IF/Prolog V5.3

Reference Manual

Siemens AG Austria

Is there

anything you would like to tell us about this manual?
Please send us your comments.

Siemens AG Austria
PSE KB B3
Gudrunstrasse 11
A-1100 Vienna
Austria

Fax.: +43-1-1707 56992

email: prolog@siemens.at

The information in this document is subject to change and does not represent a commitment on the part of Siemens AG Austria. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open and the X device are trademarks of X/Open Company Ltd.

Copyright ©Siemens AG Austria, 1999. All rights reserved.
The reproduction, transmission, translation or exploitation of this document or its contents is not permitted without express written authority. Offenders will be liable for damages. Delivery subject to availability; right of technical modifications reserved.

Contents

Contents iii

1 Introduction 1

2 Reference section on commands 5

`link_prolog` – Reconfigure Prolog 6

`procmp` – Invoke the Prolog compiler 7

`procref` – Invoke the Prolog definition referencer 8

`prolog` – Start of the Prolog system 9

`proopt` – Invoke the Prolog optimizer 13

`propp` – Invoke the Prolog preprocessor 14

`protags` – Invoke the Prolog tag generator 16

`proxref` – Invoke the Prolog cross-referencer 17

3 Overview by functionality of predicates 19

4 Reference section for the predefined predicates 37

`!/0` – Disable backtracking 39

`,/2` – Conjunction of goals 42

`-->/2` – Create a grammar rule 44

`->/2` – IF-THEN conjunction of goals 47

`./2` – List notation for consult/1, reconsult/1 50

`:/2` – Specify the module of the called predicate 52

`;/2` – Disjunction of goals 53

`</2, >/2, =</2, >=/2, =:=/2, =\=/2` – Compare arithmetic expressions . 55

=/2 – Unify terms . . . 58
=../2 – Convert between a structure and a list . . . 60
==/2, \==/2 – Compare terms for identity . . . 63
@</2, @=</2, @>/2, @>=/2, @=/2, @\=/2 – Compare terms . . . 65
\=/2 – Test for non-unifiability . . . 68
@/2 – Specify module context for predicate call . . . 70
\+/1 – Negation through failure . . . 71
abolish/1 – Remove a predicate from the database . . . 73
abort/0 – Return to the toplevel loop of IF/Prolog . . . 75
alarm/1 – Schedule alarm . . . 76
append/3 – Append or decompose lists . . . 77
arg/3 – Access individual arguments of a structure . . . 80
array/1 – Test for array . . . 82
array/2 – Query dimension of an array . . . 83
asserta/1/2 – Insert a clause in the database . . . 84
asserta_with_names/2/3 – Insert a clause in the database . . . 86
assertz/1/2 – Insert a clause in the database . . . 88
assertz_with_names/2/3 – Insert a clause in the database . . . 90
assign_alias/2 – Define an alias for an input/output stream . . . 92
at_end_of_line/0/1 – Query end of line . . . 93
at_end_of_stream/0/1 – Query end of stream . . . 95
atom/1 – Test for atom . . . 97
atom_chars/2 – Convert between an atom and individual characters . . . 99
atom_codes/2 – Convert between an atom and individual character codes . . . 101
atom_concat/3 – Compose or decompose an atom . . . 103
atom_length/2 – Count the characters in an atom . . . 105
atom_number/2 – Convert between an atom and a number . . . 107
atom_part/4 – Determine the subatom of an atom . . . 109
atom_prefix/3 – Determine the prefix of an atom . . . 111
atom_split/3 – Decompose an atom . . . 113

atom_suffix/3 – Determine the suffix of an atom . . . 115
atomic/1 – Test for constant . . . 117
bagof/3 – Find selected solutions of a goal . . . 118
begin_module/1 – Define the body of a module . . . 121
break/0 – Generate a new input loop . . . 123
byte_length/2 – Count the bytes in an atom . . . 124
call/1 – Call a goal . . . 125
call/2 – Call a goal . . . 127
calling_context/1 – Query the module argument of metapredicates . . . 129
cancel_alias/1 – Cancel an alias for an input/output stream . . . 130
catch/3 – Intercept a context jump . . . 131
char_code/2 – Convert between a character and a character code . . . 133
char_conversion/2 – Define a character conversion . . . 135
char_conversion/2 – Define a character conversion (directive) . . . 136
chdir/1 – Change current directory . . . 138
clause/2 – Search the database for specific clauses . . . 140
clause_with_names/3 – Search the database for specific clauses . . . 142
clear_eol/0/1 – Clear to end of line . . . 144
clear_eos/0/1 – Clear to end of screen . . . 146
clear_screen/0/1 – Clear screen . . . 148
close/1/2 – Close an input/output stream . . . 150
compare/3 – Compare terms . . . 152
compile/1 – Compile a Prolog text . . . 155
compound/1 – Test for structure . . . 157
concat_atom/2/3 – Concatenate individual atoms to form an atom . . . 159
connect/3 – Connect a head and a tail to form a list . . . 161
consult/1 – Read in a Prolog file . . . 163
context/2 – Create a context . . . 166
copy_term/2 – Create a term copy . . . 170
create_array/2 – Create an array . . . 171

current_alias/2 – Query aliases of input/output streams 172
current_char_conversion/2 – Information on character conversions 174
current_default_module/1 – Query current module 175
current_device_control/3 – Query information on device drivers 176
current_error/1 – Query current error output stream 178
current_global/1 – Query global variables 179
current_host/1 – Query names of current host 180
current_input/1 – Query current input stream 181
current_language/1 – Query current language 182
current_memory_management/3 – Query memory management parameters . 183
current_module/1 – Query modules 184
current_op/3 – Information on operators 186
current_output/1 – Query current output stream 189
current_predicate/1 – Query visible predicates 190
current_prolog_flag/2 – Query Prolog flags 191
current_signal/2 – Query signals and modes 197
current_socket/3 – Query communication sockets 199
current_stream_control/3 – Query information on device drivers 200
current_visible/2 – Query visible predicates 202
cyclic/1 – Test for cyclic term 204
debug/1 – Activate the debugger for a goal 205
debug_begin/0 – User-defined debugger initialization 207
debug_config/3 – Configure the debugger 208
debug_end/0 – User-defined termination of the debugger 211
debug_mode/3 – Query and set the debug mode of a predicate 212
default_editor/2 – Query and set the editor 215
device_control/2 – Control an input/output device 217
digit/1 – Test for digit 218
discontiguous/1 – Declare discontiguous predicate definition 219
dynamic/1 – Declare predicates as modifiable 221

edit/0/1 – Edit a file . . . 223
end_module/0/1 – Conclude the definition of a module . . . 225
ensure_loaded/1 – Insert a Prolog file in the source text once . . . 227
error_tell/1 – Set current error output stream . . . 229
error_telling/1 – Query current error output stream . . . 231
error_told/0 – Reset current error output stream . . . 232
exception/2 – Raise an exception . . . 233
exception_handler/3 – Define local exception handling . . . 236
exec/1/2 – Overlay process . . . 238
expand_term/2 – Transform a grammar rule . . . 240
export/1 – Export predicates . . . 241
fail/0 – Initiate backtracking . . . 243
file_test/2 – Check access permission for a file . . . 245
filepos/2/3 – Determine position within an input stream . . . 247
findall/3 – Find all solutions of a goal . . . 249
float/1 – Test for floating-point number . . . 252
float_format/2 – Query and set floating-point format . . . 253
flush_output/0/1 – Flush the buffer of an output stream . . . 256
for/3 – Generate a sequence of integers . . . 258
functor/3 – Generate or analyze a structure . . . 261
garbage_collection/1 – Activate garbage collection . . . 264
get_array/3 – Query array element . . . 265
get_byte/1/2 – Input a byte . . . 266
get_char/1/2 – Input a character . . . 268
get_code/1/2 – Input a character . . . 271
get_file_info/2/3 – Get operating system information on files . . . 273
get_global/2 – Query the value of a global variable . . . 276
get_last/2/3 – Determine the cursor position . . . 277
get_socket_option/3 – Get socket option . . . 279
get_until/3/4 – Read up to a specific character . . . 281

getchar/3 – Access a character in an atom 283
getcwd/1 – Query current working directory 285
getenv/2 – Query environment variable 287
ground/1 – Test for ground instantiation 289
halt/0/1 – Terminate Prolog 290
help/0/1 – Display help information 291
host_addr/2 – Query names or addresses of host 294
if/3 – IF-THEN-ELSE conjunction of goals 296
import/1/2 – Import predicates 299
include/1 – Include a Prolog file in the source text 302
index/3 – Determine the position of a character string in an atom 304
init/1,initialization/1 – Initialization goal 306
integer/1 – Test for integer 308
is/2 – Evaluate arithmetic expressions 309
letter/1 – Test for letter 316
list_last/2 – Last element in list 317
list_length/2 – Count the elements in a list 319
list_mode/3 – Query and set the list mode of a predicate 321
list_nth/3 – Access elements of a list 324
listing/0/1 – List the clauses in the database 326
load/1 – Load compiled Prolog text 328
load_system/1 – Restore the old system status 330
localtime/9 – Determine the date and time 332
lower_upper/2 – Convert between lowercase and uppercase letters 334
manual/0/1 – Output manual pages 336
match/2/3 – Pattern matching 338
match_atom/2 – Concatenate matching atoms to form an atom 341
member/2 – List membership 344
memberchk/2 – List membership 347
meta/1 – Declare metapredicates 349

modify_mode/3 – Query and set the modify mode of a predicate 351
module/1 – Define a module 354
move_cursor/2/3 – Position the cursor 356
multifile/1 – Declare predicates as being scattered over several files 358
net_service/3 – Query communication service 360
nl/0/1 – Output newline 362
nonmember/2 – List membership 364
nonotify/0 – Do not output load message 366
nonvar/1 – Test for instantiation 367
not/1 – Negation through failure 368
number/1 – Test for number 370
number_chars/2 – Convert between a number and a list of characters 371
number_codes/2 – Convert between a number and a list of character codes . 374
once/1 – Activate a goal once only 377
op/3 – Define and delete operators 379
op/3 – Define and delete operators (directive) 383
open/3/4 – Open an input/output stream 385
outtab/1/2 – Position the cursor 389
parse_atom/6 – Parse a character string in accordance with Prolog syntax . 392
peek_byte/1/2 – Input a byte 395
peek_char/1/2 – Input a character 397
peek_code/1/2 – Input a character 399
phrase/2 – Call a grammar rule 401
pop_global/2 – Query global variable and remove value 403
portray/2 – User-defined output predicate 404
predicate_mode/3 – Query and set the exception mode of a predicate . . . 406
predicate_type/2 – Query predicate type 409
print/1/2 – Output a term 411
private/1 – Declare predicates as private 413
profile/1/2 – Activate profiling for a goal 415

profile_reset/0 – Reset profiling statistics 417
profile_stat/0/1 – Query profiling statistics 418
program/0 – Automatically call a Prolog goal 420
program_parameters/1 – Query parameters from IF/Prolog call 422
prolog_serial/1 – Query IF/Prolog serial number 424
prolog_version/1 – Query IF/Prolog version 425
proroot/1 – Query path of IF/Prolog system 426
push_global/2 – Set value of a global variable 427
put_byte/1/2 – Output a byte 428
put_char/1/2 – Output a character 430
put_code/1/2 – Output a character 432
rational/1 – Test for rational number 434
rational/3 – Decompose a rational number 435
read/1/2 – Input a term 437
read_error/2/3 – Query position of a syntax error 440
read_term/2/3 – Input a term 442
reconsult/1 – Read in a Prolog file 446
reduce/3 – Reduce the depth of a structure 449
reexport/1/2 – Reexport predicates 450
regexp/2/3 – Match a regular expression 452
remove/3 – Remove a sublist from a list 454
repeat/0 – Generate a choice point 456
reset_streams/0 – Restore default values for current streams 458
retract/1/2 – Remove a clause from database 459
retract_with_names/2/3 – Remove a clause from database 461
reverse/2 – Reverse a list 463
save_module/1 – Store current module in a file 465
save_runtime/1 – Save system status for creating applications 467
save_system/1 – Save current system status 469
see/1 – Set current input stream 471

seeing/1 – Query current input stream . . . 473
seen/0 – Reset current input stream . . . 474
select/3/4/5 – Synchronous input multiplexing . . . 475
set_array/3 – Set array element . . . 477
set_default_module/1 – Set current module . . . 479
set_error/1 – Set current error output stream . . . 480
set_global/2 – Set value of a global variable . . . 482
set_input/1 – Set current input stream . . . 483
set_memory_management/3 – Set memory management parameters . . . 484
set_output/1 – Set current output stream . . . 487
set_prolog_flag/2 – Set value of a Prolog flag . . . 488
set_prolog_flag/2 – Set value of a Prolog flag . . . 493
set_socket_option/3 – Set socket option . . . 495
set_stream_position/2 – Position within an input/output stream . . . 497
setof/3 – Find selected solutions of a goal . . . 499
signal_control/2 – Control signal processing . . . 502
signal_handler/3 – Define local signal handling . . . 505
skip_line/0/1 – Skip an input line . . . 507
socket/3 – Create communication socket . . . 509
socket_accept/3 – Accept a connection . . . 511
socket_bind/2 – Bind a name to a socket . . . 513
socket_close/1 – Close a socket . . . 515
socket_connect/2 – Connect a socket . . . 516
socket_listen/1/2 – Listen for connections . . . 518
socket_raw_receive/4/5 – Receive data from socket . . . 519
socket_receive/2/3 – Receive data from socket . . . 521
socket_send/2/3 – Send data to socket . . . 523
socket_shutdown/1/2 – Close a socket . . . 525
sort/2 – Sort a list . . . 527
statistics – System statistics . . . 529

stream_control/2 – Control an input/output stream . . . 532
stream_copy/2 – Redefine streams . . . 535
stream_device/2 – Determine device name of an input/output stream . . . 538
stream_property/2 – Query information on input/output streams . . . 540
stream_type/2 – Query type of an input/output stream . . . 543
sub_atom/5 – Analyze an atom . . . 545
syntax_error/2 – Assignment of syntax error number to error message . . . 548
system/0 – Start the operating system command interpreter . . . 550
system/1/2 – Execute an operating system command . . . 552
system/3/5 – Execute an operating system command with input and output specified . . . 554
system_hostid/1 – Query machine identification . . . 558
system_name/1 – Query operating system . . . 560
system_parameters/1 – Query system parameters from IF/Prolog call . . . 561
tab/1/2 – Move the cursor forward . . . 563
tell/1 – Set current output stream . . . 565
telling/1 – Query current output stream . . . 567
throw/1 – Initiate a context jump . . . 568
timezone/4 – Determine time and time zones . . . 570
told/0 – Reset current output stream . . . 572
trace/1 – Activate tracing for a goal . . . 573
trace_begin/0 – User-defined tracer initialization . . . 575
trace_config/3 – Configure the tracer . . . 576
trace_end/0 – User-defined termination of the tracer . . . 578
trace_output/4 – User-defined output predicate for the tracer . . . 579
true/0 – Goal which is always true . . . 580
tty_size/2/3 – Query the screen size . . . 581
unify_with_occurs_check/2 – Unify terms with occurs check . . . 583
unix_fork/1 – Spawn process . . . 584
unix_getpid/1 – Query process ID of current process . . . 587
unix_kill/2 – Send signal to a process . . . 588

unix_make_pipe/1 – Create a pipe . . . 590
unix_wait/1/2 – Wait for termination of a child process . . . 592
unload/1 – Remove module from the database . . . 594
unset_global/1 – Delete a global variable . . . 595
user_parameters/1 – Query user parameters from IF/Prolog call . . . 596
var/1 – Test for variable . . . 598
windows_chdrive/1 – Change drive . . . 599
windows_getdrive/1 – Query drive . . . 600
write/1/2 – Output a term . . . 601
write_atom/2 – Convert a term into an atom . . . 603
write_canonical/1/2 – Output a term . . . 605
write_formatted/2/3 – Formatted output of terms . . . 607
write_formatted_atom/3 – Formatted output of terms . . . 611
write_term/2/3 – Output a term . . . 613
writeq/1/2 – Output a term in readable form . . . 618
writeq_atom/2 – Convert a term into an atom . . . 620

5 Overview by functionality of debugger commands 623

6 Reference section on debugger commands 629

+/0,-/0,++/0,--/0 – Scrolling . . . 630
abort/0 – Terminate interactive debugger . . . 631
activate_stop/0/1 – Reactivate breakpoint . . . 632
back_clause/0 – Branch back to clause head . . . 634
back_parent/0 – Branch back to CALL port of parent goal . . . 635
back_subgoal/0 – Branch back to CALL port of subgoal . . . 636
break/0 – Start new input loop . . . 637
call/1 – Execute a Prolog goal . . . 638
continue/0 – Continue to next breakpoint . . . 639
deactivate_stop/0/1 – Deactivate breakpoint temporarily . . . 640
error/2 – Raise an exception . . . 642

exit/0 – Force current subgoal to succeed . . . 643
fail/0 – Force the current subgoal to fail . . . 644
fast_skip/0/1 – Accelerate execution to exit of a subsequent subgoal . . . 645
halt/0 – Terminate debugger and IF/Prolog . . . 646
help/0 – Display help information . . . 647
nonstop/0 – Continue without stopping . . . 648
port/2 – Set debugger port . . . 649
remove_stop/0/1 – Remove explicit breakpoints . . . 651
set_debug_depth/1 – Set structure depth . . . 653
set_history/1 – Configure history . . . 654
set_trace_depth/1 – Set structure depth for trace outputs . . . 655
set_trace_length/1 – Set length of trace field . . . 656
skip/0/1 – Execute up to exit of a subsequent subgoal . . . 657
snapshot/0/1/2 – Store field . . . 658
step/0 – Execute up to next monitored port . . . 660
stop/4,stop_interactive/1/2/4 – Set explicit breakpoint . . . 661
system/0 – Interrupt interactive debugger and start the shell . . . 666
trace/0/1 – Logging passes through ports . . . 667
trace_port/2 – Set trace ports . . . 668
unify/2 – Unify variable . . . 670
view_.../0 – Select display fields . . . 671
view_ancestors/0/1 – Show dynamic call chain . . . 672
view_configuration/0 – Display default values . . . 673
view_constraints/0 – Display constraints . . . 674
view_module/0/1 – Display list of predicates in module . . . 675
view_predicate/0/1 – Display listing of a predicate . . . 676
view_..._screen/0 – Display history . . . 678
view_stops/0 – Display explicit breakpoints . . . 679
view_variable/0/1 – Display variable constraints . . . 680

7 Overview by functionality of C interface 683

8 Reference section on C functions **691**

`Cboot()` – Link a C function to IF/Prolog 692

`CPRED()` – Add a complex C function to IF/Prolog 693

`CPRIM()` – Add a simple C function to IF/Prolog 694

`Cshutdown()` – Call a C function on IF/Prolog shutdown 695

`DeviceCreate()` – Register device driver 696

`EndProlog()` – Release memory space used by IF/Prolog 697

`ErrorContextClear()` – Clear error context 698

`ErrorContextGet()` – Query error context 700

`ErrorContextIsSet()` – Status of the error context 701

`ErrorContextSet()` – Set error context 703

`fprintt()` – Formatted output of Prolog terms and C data objects 706

`InitProlog()` – Initialize IF/Prolog and define the standard media 707

`MCPRED()` – Add a complex C function to a module 708

`MCPRIM()` – Add a simple C function to a module 709

`outputlen()` – Determine length of formatted output 710

`ParseError()` – Assignment of syntax error number to syntax error message 711

`ParseProlog()` – Parse a goal string and store the information 713

`printt()` – Formatted output of Prolog terms and C data objects 715

`PrologArg()` – Access predicate arguments 716

`PrologClose()` – Deactivate the current Prolog goal 718

`PrologEnableEpilog()` – Enable epilog 719

`PrologError()` – Query error 721

`PrologFetch()` – Find solutions for a goal 722

`PrologGoal()` – Prolog goal as term 723

`PrologIsFloat()` – Get floating-point number from variable 724

`PrologIsInteger()` – Get integer from variable 725

`PrologIsString()` – Get character string from variable 726

`PrologIsTerm()` – Get Prolog term from variable 727

`PrologModule()` – Get Prolog module 728

PrologOpen() – Create current Prolog goal 729
PrologUndo() – Undo unification 730
QueryDispose() – Release passive Prolog goal 733
QueryOpen() – Activate a passive Prolog goal 734
sprintt() – Formatted output of Prolog terms and C data objects 735
StreamFlush() – Flush Prolog output buffer 736
StreamMode() – Query stream mode 737
TermAddList() – Add to a Prolog list 738
TermArg() – Access structure arguments 740
TermCloseList() – Close a Prolog list 741
TermCollect(),TermCollectSequence() – Release term variables 742
TermContext() – Determine current term context 745
TermDecompose() – Classify term 746
TermIsAtom() – Test for atom 749
TermIsCompound() – Test for structure 750
TermIsFloat() – Test for floating-point number 752
TermIsFloatExpression() – Test for floating-point expression 753
TermIsFunctor() – Test for structure 754
TermIsInteger() – Test for integer 756
TermIsIntegerExpression() – Test for integer expression 757
TermIsList() – Test for list 758
TermIsNil() – Test for empty list 760
TermIsUniversal() – Test for structure 761
TermIsVar() – Test for variable 763
TermMakeAtom() – Generate a Prolog atom 764
TermMakeCompound() – Generate a Prolog structure 765
TermMakeFloat() – Generate a Prolog floating-point number 767
TermMakeFunctor() – Generate a Prolog structure 768
TermMakeInteger() – Generate a Prolog integer 770
TermMakeList() – Generate a Prolog list 771

`TermMakeNil()` – Generate an empty list . . . 773
`TermMakeUniversal()` – Generate a Prolog structure . . . 774
`TermMakeVar()` – Generate a Prolog variable . . . 776
`TermOpenList()` – Generate a Prolog list . . . 777
`TermType()` – Classify term . . . 778
`TermUnify()` – Unify terms . . . 780
`TermUnifyAtom()` – Unify term with atom . . . 781
`TermUnifyCompound()` – Unify term with structure . . . 782
`TermUnifyFloat()` – Unify with floating-point number . . . 784
`TermUnifyFunctor()` – Unify term with structure . . . 785
`TermUnifyInteger()` – Unify term with integer . . . 787
`TermUnifyList()` – Unify term with list . . . 788
`TermUnifyNil()` – Unify term with the empty list . . . 789
`TermUnifyUniversal()` – Unify term with structure . . . 790
`TermUnifyVar()` – Unify terms . . . 792

9 Prolog syntax **793**
9.1 Basic Prolog elements . . . 793
9.2 Terms . . . 798
9.2.1 Constants . . . 799
9.2.2 Variables . . . 804
9.2.3 Structures . . . 808
9.3 Operators . . . 813
9.3.1 Converting expressions into equivalent normal structures . . . 813
9.3.2 List of built-in operators . . . 818
9.3.3 Test and definition predicates . . . 820
9.4 Arithmetic expressions and functions . . . 820
9.5 Prolog text . . . 821

A ISO Standard Directives and Predicates **825**

B ASCII table **827**

C Glossary **831**

Bibliography **835**

Index **836**

Chapter 1

Introduction

The IF/Prolog system from Siemens AG Austria is an implementation of the ISO Prolog standard (ISO = International Standardization Organization). This standard was prepared by ISO Working Group 17, comprising representatives from various national standardization bodies.

IF/Prolog also contains interfaces and predicates which extend the language and ensure compatibility with earlier versions of the product.

The Standard dictates us to supply a strictly conforming mode, where the Prolog system only accepts and supplies conforming language predciates. To invoke this mode, see the section on invoking IF/Prolog in the User's Guide.

The *IF/Prolog Reference Manual* documents the language features of Prolog.

Target group

The *IF/Prolog Reference Manual* is intended for anyone working with Prolog and writing Prolog programs.

The manual is not designed as a textbook for the Prolog language. You should therefore already have a basic knowledge of Prolog. Suitable textbooks ([12], [13], [14]) are given in the *Bibliography*.

You can develop a feel for Prolog by working through the sample session described in the opening section of the *IF/Prolog User's Guide* [2]. You can do this without any previous knowledge of the language.

The documentation for IF/Prolog comprises of the following manuals:

- *IF/Prolog Reference Manual*
- *IF/Prolog User's Guide*
- *IF/Prolog Windows Interfaces*

- *IF/Prolog OSF/Motif Interface*
- *IF/Prolog Constraints Package*
- *IF/Prolog Java Interface*

The *IF/Prolog Reference Manual* contains a description of the semantics, built-in predicates, C interface functions, debugger commands and environment control of IF/Prolog. It also contains an overview of the syntax of the Prolog language.

Predicates associated with an interface or package are described in the respective manuals. The *IF/Prolog User's Guide* describes how to work with Prolog on a computer running under the UNIX, DOS, Windows and other operating system.

As the set of manuals for IF/Prolog are not tutorials, you should be familiar with the Prolog language. You should also be familiar with the basics of the operating system and know how to use one of the editors installed on your computer.

The following notational conventions are used throughout this manual:

xxx	Syntax definitions are enclosed within a frame.
`bind`	Prolog language elements, operating system commands and outputs from the system are printed in `teletype` font.
Name	*Italics* are used to represent variable parts in inputs and outputs where you should substitute them with your own values.
[ ]	Square brackets denote optional entries in the syntax notation; the brackets are not part of the Prolog text.
[ ]	Square brackets in bold type are elements of the Prolog list notation and are part of the Prolog text.
{ }	Braces denote alternatives in the syntax notation; the brackets are not part of the Prolog text.
|	A bar denotes alternatives in the syntax notation.
( )	Parentheses are required parts of the Prolog predicate notation and is part of the Prolog text.
...	Ellipsis indicate that the preceding syntax element may be repeated.
`atom/1`	Predicates are specified in the form *Name*/*Arity*.
`nl/0/1`	Several predicates with the same name and different arities are denoted in this form.
[1]	A number in square brackets indicates a reference to another manual or a textbook. The number identifies the publication in the Bibliography at the end of the manual.

The following pictograms are also used:

i for important advice and related information.

! for warnings.

Chapter 2

Reference section on commands

This section contains descriptions of the commands in alphabetical order.

Reconfigure Prolog

```
link_prolog [name]
```

The command `link_prolog` creates a new Prolog system kernel from the Prolog base system and additional Prolog packages. Inclusion of these packages is optional and can be controlled by the user. If the argument *name* is given on the command line, the new Prolog kernel is stored in file *name*, otherwise the new Prolog kernel is stored under the name `prostd`. If *name* does not represent a path name the new Prolog kernel is stored in the *bin* subdirectory of the Prolog system.

The following steps are required to link a new Prolog kernel:

- The command `link_prolog` *name* is called. This command can be found in the directory *$PROROOT/c* of the Prolog system (e.g. */opt/lib/prolog* on some UNIX systems)
- For each package supported by the Prolog system (**Informix**, **Motif**, **Constraints**) that has been installed, the system will ask if it should be included in the system kernel:

  ```
  Do you wish to use ...  ?
  ```

  If another software product required by a package (e.g. **Informix** or **Motif**) has not been installed, a warning message is printed and this package is not included.
- If a Prolog kernel *name* already exists, the user is asked for confirmation to overwrite the old one. If the user does not want to overwrite the kernel, the system asks the user for a new name.
- The new kernel is linked. A success or failure message is printed.

Exceptions

IF/Prolog Kernel *name* **already exists.**
: There is already a file called *name*.

You have to install ...
: A product named in the message has to be installed before it can be linked to the Prolog kernel.

(Re-)linking of IF/Prolog Kernel *name* **failed**
: The new Prolog kernel could not be linked.

See also

prolog

Invoke the Prolog compiler

`procmp [`*Option* **...**`]` *Filename*

The command `procmp` compiles the Prolog text contained in the file *Filename*. The name of the compiled Prolog file is *Name.cmp*. *Name* is generated from the filename *Filename* by removing a possible suffix *.pro*. If a file *Name.cmp* already exists, it will be overwritten.

After a successful compilation the system invokes the command `proopt` to optimize the generated code. The contents of the file *Name.cmp* is similar to a saved system state (see `save_system/1`, `save_module/1`) and can be loaded with the command line option `-l` or the predicate `load/1`.

After a successful compilation the compiler exits with the exit value 0. If an error occurs a message is printed and the compiler exits with the exit value 1. In this case no result file is created.

The command `procmp` processes some command line options which are also valid for the command `prolog`:

`-c` *File*

`-l` *File* These options can be used to provide module interfaces of modules from which predicates are imported.

`-sp` *Path* This option determines the search path of files. It is used as search path for files to be loaded or included (see `include/1`).

`-nowarnings`
Suppress the output of possible warning messages.

`-iso` Only the features required by the ISO standard are available. The implementation-specific extensions of IF/Prolog are not available.

Exceptions

All the error messages and warnings from `read_term/2`, `assertz/1` and `assertz_with_names/2` may occur.

See also

prolog, proopt, compile/1, save_system/1, save_module/1

Invoke the Prolog definition referencer

`procref [`*Flag* **...**`]` *Filename* `[` *Filename* **...** `]`

The command `procref` generates a list of defined predicates with declaration information from Prolog source files and outputs the list on standard output.

If the file *Filename* exists and is readable, this is the input file. Otherwise, if *Filename* does not already have a *.pro* or *.ppp* suffix, the suffix *.pro* will be appended to *Filename*. This is then the name of input file.

The following flags are allowed on the command line:

`-a`	the output is in alphabetical order (default)
`-u`	the output is in the order of occurrence in files
`-e`	only exported predicates are output
`-f`	output line is not limited to 80 characters

Exceptions

Prolog exceptions from file access (open, read) may occur.

existence error(source_sink)
The input file does not exist.

permission error(open, source_sink)
The input file cannot be opened for reading.

Compatibility

V5.1A The command `procref` is new.

Start of the Prolog system

prolog [*Option* **...**] [`--`] [*Parameter* **...**]

The command `prolog` starts the Prolog system using the system parameters given in *Option*.

The user parameters given in *Parameter* are not processed by the Prolog system. A list of the user arguments specified in the call can be queried by means of `user_parameters/1`. If after initialization a predicate `program/0` is visible, then this predicate is called. Once it has been executed, the Prolog system is exited with the exit value 0.

If the predicate `program/0` is not visible, the Prolog input loop (`break/0`) is started. The Prolog session can then be terminated with the call of `halt/0/1`.

The meanings of the various system arguments are explained in the following.

`-root` *pathname*

Specify pathname under which the files of the Prolog system are installed. This option takes precedence over the pathname given during installation or the environment variable `PROROOT`.

If this parameter is not given, the pathname defined by the environment variable `PROROOT` is used. If the environment variable `PROROOT` is not set, the pathname given during installation is used.

`-r` *file* Load system status from *file* (see `save_system/1`). If *file* cannot be opened or if *file* does not have the suffix *.pst*, the suffix is appended and the name thus produced is used.

Independently of its position in the command line, this parameter will be evaluated before the system parameters `-st`, `-c` and `-l`.

`-st` *file* Specify which startup file, if any, should be loaded or consulted. Startup files are loaded or consulted before other files are loaded (`-l`) or consulted (`-c`), but after a system state has been loaded (`-r`).

If a hyphen (`-`) is specified for *file*, then no start file is loaded or consulted.

If the filename *file* has the suffix *.pro*, the file is consulted. If it has the suffix *.cmp*, it is loaded. Otherwise the system tries to consult the file *file*. If the file *file* does not exist, the system tries to append the suffixes *.pro* and *.cmp* and uses the filenames thus produced.

If this system parameter is omitted and the environment variable `PRORC` is set, the system acts as if `-st $PRORC` was given. In all other cases, the system acts as if `-st` *$HOME/.prorc* was given.

`-c` *file* Consult *file* (see `consult/1`). If the file *file* does not exist and if *file* does not have the suffix *.pro*, the suffix is appended and the name thus produced is used.

`-l` *file* Load *file* (see `load/1`). If it does not exist and if *file* does not have the suffix *.cmp*, the suffix is appended and the name thus produced is used.

The system parameters `-c` and `-l` are processed in the order of their appearance in the command line. The system parameters `-c` and `-l` are processed after the parameters `-r` and `-st`.

Independently of their position in the command line, the following system parameters are evaluated **before** the parameters `-r`, `-st`, `-c` and `-l`.

`-sp` *search_path*
Set the initial value of the Prolog `search_path` flag to the list specified in *search_path*. If more than one path name is given, the pathnames are separated by means of a system dependent character like the colon `:` (UNIX) or the semicolon `;` (DOS).

`-msg` *number*`[k|K]`
Set the initial size of the global stack to *number* bytes. If `k` or `K` after *number* is given, it is set to *number* kilobytes. If required for program execution, the memory area is automatically increased.

`-msc` *number*`[k|K]`
Set the initial size of the constraint data stack to *number* bytes. If `k` or `K` after *number* is given, it is set to *number* kilobytes. If required for program execution, the memory area is automatically increased.

`-mst` *number*`[k|K]`
Set the initial size of the variable instantiations stack (trail) to *number* bytes. If `k` or `K` after *number* is given, it is set to *number* kilobytes. If required for program execution, the memory area is automatically increased.

`-msl` *number*`[k|K]`
Set the initial size of the local data stack to *number* bytes. If `k` or `K` after *number* is given, it is set to *number* kilobytes. If required for program execution, the memory area is automatically increased.

`-msx` *number*`[k|K]`
Set the initial size of the extended terms stack to *number* bytes. If `k` or `K` after *number* is given, it is set to *number* kilobytes. If required for program execution, the memory area is automatically increased.

`-ms` *number*`[k|K]`
Set the initial size of the total memory of the Prolog system to *number* bytes. If `k` or `K` after *number* is given, it is set to *number* kilobytes. If required for program execution, the memory area is automatically increased.

`-mi` *number*`[k|K]`
Set the minimum size of the memory blocks requested by the Prolog system to *number* bytes. If `k` or `K` after *number* is given, it is set to *number* kilobytes.

`-debug` The initial value of the Prolog flag `debug` is set to `on`. This means that all the predicates which you read into the database with `consult/1`, `reconsult/1` or `assert/1/2` are implicitly declared as dynamic, thereby supporting the debugging of a program.

`-iso` Only the features required by the ISO standard are available. The implementation-specific extensions of IF/Prolog are not available.

`-nobuf` The standard input streams are not buffered by Prolog.

`-nocopyright`
The copyright message is not output when Prolog starts.

`-nonotify`
The initial value of the Prolog flag `consult` is set to `nonotify`. This means that no messages are displayed on the screen when a file is loaded.

`-nosignal`
Prevents the Prolog system from managing signals itself, i.e. no signal handling routines are defined. The predicate `signal_control/2` has no effect.

`-notty` The standard streams `screen` and `keyboard` are not linked to the terminal, but are mapped to the standard input and standard output. In this case it is not possible to use the keyboard and the screen as the input/output devices when the standard input/output has been redirected.

`-nowarnings`
The initial value of the Prolog flag `warnings` is set to `off`.

`-prompt` The initial value of the Prolog flag `prompt` is set to `on`.

`-stream` Prolog acts as if no terminal is available. The debugger is started without a graphical user interface (see also `debug_config/3`).

`-verbose` A message containing configuration and license information and creation date of IF/Prolog is output when Prolog starts.

`-xenv` If a OSF/Motif environment (X server) is available, the Prolog system starts the graphical development environment under OSF/Motif.

`--` Identifies the end of the system arguments. All following arguments are considered by the Prolog system to be user arguments, even if the name is the same as that of a system argument.

The end of the system arguments is also considered to have been reached if the command line contains an argument which is not a system argument.

Exceptions

Prolog license/keyinformation missing
Prolog can not be started because no license information could be found.

could not initialize PROLOG
Prolog could not be initialized.

illegal size for *Option*
The value for *Option* is not legal.

Parameter '-root' not set correctly?
The parameter *pathname* in the option `-root` *pathname* is not the name of the valid Prolog system directory or this directory cannot be accessed.

environment variable PROROOT not set correctly?
The path name defined by the environment variable `PROROOT` is not the name of the valid Prolog system directory or the directory cannot be accessed.

Parameter '-r' not set correctly?
The file *file* selected by the option `-r` *file* is not a valid save state.

See also

program_parameters/1, system_parameters/1, user_parameters/1, current_prolog_flag/2

Invoke the Prolog optimizer

`proopt` *Filename*

The command `proopt` optimizes the system state saved in the file *Filename* by optimizing all static user defined predicates. The saved system state in the file *Filename* will be modified. A saved system state can be generated with the predicates `compile/1`, `save_module/1`, `save_system/1` or `save_runtime/1`.

Optimizing an already optimized system state has no effect. The save state generated by the command `procmp` is already optimized.

The size of the file can increase by optimization. This does not mean that the runtime memory requirement will also increase.

After a successful optimization the optimizer exits with the exit value 0. If an error occurs a message is printed and the optimizer exits with the exit value 1.

Exceptions

illegal savestate
: The file *Filename* is not a legal save state.

not enough memory
: The optimizer does not get enough dynamic memory by the operating system.

could not open file
: The file *Filename* does not exist or cannot be opened for reading and writing.

could not open temporary file
: A temporary file could not be opened.

write error
: An error occurred during writing of the file. This file is now inconsistent and will therefore be removed.

read error
: An error occurred during reading a file.

usage
: `proopt` was called without or with more than one parameters.

See also

procmp, compile/1, save_module/1, save_system/1

Invoke the Prolog preprocessor

`propp [`*Flag* **...**`]` *Filename*

The command `propp` preprocesses a Prolog file that may contain conditional statements in the style of the C preprocessor. The input file contains Prolog code plus conditional expressions. The output file will only contain those parts of the Prolog code that remain after evaluation of the conditional expressions.

If the file *Filename* exists and is readable, this is the input file. Otherwise, if *Filename* does not already have a *.ppp* suffix, the suffix *.ppp* will be appended to *Filename*. This is then the name of input file.

The name of the output file will have the suffix *.pro*. If the name of the input file does have the suffix *.ppp*, *.pro* will replace *.ppp*. Otherwise *.pro* will be appended to the name of the input file. Please note that prior contents of the output file will be overwritten.

After successful preprocessing the preprocessor exits with the exit value 0. If an error occurs a message is printed and the preprocessor exits with the exit value 1.

The following flags are allowed on the command line:

`-D`*atom*

atom is defined as an atom to be used in conditional expressions in the input file.

The following directives are allowed in conditional statements. The directives must begin at the first column of the input file.

```
#if <expression>
#ifdef <atom>
#ifndef <atom>
#elif <expression>
#else
#endif
```

The syntax of expressions is:

```
<expression> ::=
( <expression> )
! <expression>
<expression> && <expression>
<expression> || <expression>
defined(<atom>)
undefined(<atom>)
```

Exceptions

Prolog exceptions from file access (open, read, write) may occur.

existence error(source_sink)
The input file does not exist.

permission error(open, source_sink)
The input file cannot be opened for reading.
The output file cannot be opened for writing.

domain_error(preprocessor_flag, *Arg*)
An illegal preprocessor flag *Arg* occurred on the command line.

domain_error(preprocessor_expression, *Directive*)
An illegal preprocessor directive *Directive* was encountered.

domain_error(expression, *Expr*)
The preprocessor expression *Expr* is malformed.

type_error(atom, *Name*)
Only atoms are allowed in preprocessor expressions.

Compatibility

V5.1A The command `propp` is new.

Invoke the Prolog tag generator

`protags [`*Flag* **...**`]` *Filename* `[` *Filename* **...** `]`

The command `protags` generates a list of tags from Prolog source files and outputs the list on standard output. The output can be used as a **tags** file. A tags file gives the locations of specified objects (in this case predicates) in a group of files. Each line of the tags file contains the object name, the file in which it is defined, and an address specification for the object definition. A tags file can be used by some editors (UNIX: `vi` and `ex`) to quickly find these objects.

If the file *Filename* exists and is readable, this is the input file. Otherwise, if *Filename* does not already have a *.pro* or *.ppp* suffix, the suffix *.pro* will be appended to *Filename*. This is then the name of input file.

The following flags are allowed on the command line:

`-a` the output consists of functor name, filename and search pattern (default).

`-x` output predicate name, arity, line number of first occurrence and file name. This output can be used as a readable predicate index.

Exceptions

Prolog exceptions from file access (open, read) may occur.

existence error(source_sink)
The input file does not exist.

permission error(open, source_sink)
The input file cannot be opened for reading.

Compatibility

V5.1A The command `protags` is new.

Invoke the Prolog cross-referencer

`proxref [`*Flag* **...**`]` *Filename* `[` *Filename* **...** `]`

The command `proxref` generates a list of predicates with their cross-references from Prolog source files and outputs the list on standard output.

The list of cross-references is either forward- or backward-chained. A **forward-chained reference** lists all predicates that are called in the body.
A **backward-chained reference** lists from which predicate(s) a predicate is called.

If the file *Filename* exists and is readable, this is the input file. Otherwise, if *Filename* does not already have a *.pro* or *.ppp* suffix, the suffix *.pro* will be appended to *Filename*. This is then the name of input file.

The following flags are allowed on the command line:

`-a`	the output is in alphabetical order (default)
`-u`	the output is in the order of occurrence in files
`-f`	generate a forward-chained reference list (default)
`-p`	generate a forward-chained reference list including builtin predicates
`-b`	generate a backward-chained reference list

Exceptions

Prolog exceptions from file access (open, read) may occur.

existence error(source_sink)
The input file does not exist.

permission error(open, source_sink)
The input file cannot be opened for reading.

Compatibility

V5.1A The command `proxref` is new.

Chapter 3

Overview by functionality of predicates

This section contains an overview of the built-in predicates, arranged into groups according to their functionality.

It also includes the language constructs, directives and special user defined predicates that can be used in the Prolog system.

The functor of each predicate is printed in bold type. Each argument is preceded by one of the characters '`@`', '`+`', '`?`' or '`-`', symbolizing the call type of the argument (see page 37).

Predicates which allow **backtracking** are identified by the character '#' before the functor.

Metapredicates are identified by the character '@' before the functor.

Directives are identified by ':-' before the functor.

Predicates with functors that are defined as operators are shown in operator notation by default.

Term classification

array(@TestTerm)	Test for array
array(@Array, ?Dimension)	Query dimension of an array
atom(@TestTerm)	Test for atom
atomic(@TestTerm)	Test for constant
compound(@TestTerm)	Test for structure
cyclic(@TestTerm)	Test for cyclic term
digit(@TestTerm)	Test for digit
float(@TestTerm)	Test for floating-point number
ground(@TestTerm)	Test for ground instantiation
integer(@TestTerm)	Test for integer

letter(@TestTerm)	Test for letter
nonvar(@TestTerm)	Test for instantiation
number(@TestTerm)	Test for number
rational(@TestTerm)	Test for rational number
var(@TestTerm)	Test for variable

Term comparison

?Term1 = ?Term2	Unify terms
@Term1 == @Term2	Compare terms for identity
@Term1 \== @Term2	Compare terms for unidentity
@Term1 **@<** @Term2	Term1 less than Term2 (order of terms)
@Term1 **@>** @Term2	Term1 greater than Term2 (order of terms)
@Term1 **@=<** @Term2	Term1 less or equal Term2 (order of terms)
@Term1 **@>=** @Term2	Term1 greater or equal Term2 (order of terms)
@Term1 **@=** @Term2	Term1 equal Term2 (order of terms)
@Term1 **@\=** @Term2	Term1 not equal Term2 (order of terms)
@Term1 \= @Term2	Test for non-unifiability
compare(?Op, @Term1, @Term2)	Compare terms
unify_with_occurs_check(?Term1, ?Term2)	Unify terms with occurs check

Term conversion

+Structure **=..** ?List	Convert a structure into a list
-Structure **=..** +List	Convert a list into a structure
arg(+Position, +Structure, ?Argument)	Access individual arguments of a structure
atom_chars(+Atom, ?List)	Decompose an atom into a list of characters
atom_chars(-Atom, @List)	Convert a list of characters into an atom
atom_codes(+Atom, ?List)	Decompose an atom into a list of character codes
atom_codes(-Atom, @List)	Convert a list of character codes into an atom
atom_number(+Atom, ?Number)	Convert an atom into a number
atom_number(-Atom, @Number)	Convert a number into an atom
char_code(+Character, ?CharCode)	Convert a character into a character code
char_code(-Character, +CharCode)	Convert a character code into a character
copy_term(@Term1, ?Term2)	Create a term copy

create_array(?Array, @Dimension)	Create an array
functor(-Structure, +Functor, +Arity)	Generate a structure
functor(@Structure, ?Functor, ?Arity)	Analyze a structure
get_array(@Array, @Index, ?Term)	Query array element
number_chars(+Number, ?List)	Decompose a number into a list of characters
number_chars(-Number, @List)	Convert a list of characters into a number
number_codes(+Number, ?List)	Decompose a number into a list of character codes
number_codes(-Number, @List)	Convert a list of character codes into a number
parse_atom(+String, +StartPosition, ?EndPosition, ?Term, ?VarList, ?Error)	Parse a character string in accordance with Prolog syntax
rational(+Number, ?Numerator, ?Denominator)	Decompose a rational number
reduce(+Level, @Term, ?ReducedTerm)	Reduce the depth of a structure
set_array(@Array, @Index, @Term)	Set array element
write_atom(@Term, ?Atom)	Convert a term into an atom
write_formatted_atom(?Atom, +Format, @TermList)	Formatted output of terms
writeq_atom(@Term, ?Atom)	Convert a term into an atom

String processing

atom_chars(+Atom, ?List)	Decompose an atom into a list of characters
atom_chars(-Atom, @List)	Convert a list of characters into an atom
atom_codes(+Atom, ?List)	Decompose an atom into a list of character codes
atom_codes(-Atom, @List)	Convert a list of character codes into an atom
#**atom_concat**(?Atom1, ?Atom2, +Atom3)	Decompose an atom
#**atom_concat**(+Atom1, +Atom2, -Atom3)	Compose an atom
atom_length(+Atom, ?Length)	Count the characters in an atom
atom_number(+Atom, ?Number)	Convert an atom into a number
atom_number(-Atom, @Number)	Convert a number into an atom
atom_part(+Atom, +Position, +Length, ?SubAtom)	Determine the subatom of an atom
atom_prefix(+Atom, +Length, ?Prefix)	Determine the prefix of an atom

atom_split(+Atom, +Delimiter, ?Subatoms)	Decompose an atom
atom_suffix(+Atom, +Length, ?Suffix)	Determine the suffix of an atom
byte_length(+Atom, ?Length)	Count the bytes in an atom
concat_atom(@List, ?Atom)	Concatenate individual atoms to form an atom
concat_atom(@List, +Delimiter, ?Atom)	Concatenate individual atoms to form an atom with separators
current_language(?Language)	Query current language
getchar(+Atom, +Position, ?Character)	Access a character in an atom
index(+Atom, +String, ?Position)	Determine the position of a character string in an atom
lower_upper(+Lowercase, ?Uppercase)	Convert lowercase letters into uppercase
lower_upper(-Lowercase, +Uppercase)	Convert uppercase letters into lowercase
match(+Mask, +Atom)	Pattern matching
match(+Mask, +Atom, ?Replacements)	Pattern matching
match_atom(@List, ?Atom)	Concatenate atoms to form an atom
#**match_atom**(+List, +Atom)	Concatenate matching atoms to form an atom
parse_atom(+String, +StartPosition, ?EndPosition, ?Term, ?VarList, ?Error)	Parse a character string in accordance with Prolog syntax
regexp(+RegExp, +Atom)	Regular expression matching
regexp(+RegExp, +Atom, ?List)	Regular expression matching
#**sub_atom**(+Atom, ?StartLength, ?Length, ?RestLength, ?SubAtom)	Analyze an atom
write_atom(@Term, ?Atom)	Convert a term into an atom
write_formatted_atom(?Atom, +Format, @TermList)	Formatted output of terms
writeq_atom(@Term, ?Atom)	Convert a term into an atom

List processing

#**append**(?Head, ?Tail, ?List)	Append or decompose lists
atom_chars(+Atom, ?List)	Decompose an atom into a list of characters
atom_chars(-Atom, @List)	Convert a list of characters into an atom
atom_codes(+Atom, ?List)	Decompose an atom into a list of character codes
atom_codes(-Atom, @List)	Convert a list of character codes into an atom
concat_atom(@List, ?Atom)	Concatenate individual atoms to form an atom

concat_atom(@List, +Delimiter, ?Atom)	Concatenate individual atoms to form an atom with separators
connect(?List, ?Head, ?Tail)	Connect a head and a tail to form a list
list_last(?Element, +List)	Last element in list
list_length(+List, ?Length)	Count the elements in a list
list_nth(?Position, +List, ?Element)	Access elements of a list
match_atom(@List, ?Atom)	Concatenate atoms to form an atom
#**match_atom**(+List, +Atom)	Concatenate matching atoms to form an atom
#**member**(?Element, ?List)	List membership
memberchk(?Element, ?List)	List membership
nonmember(@Element, @List)	List membership
number_chars(+Number, ?List)	Decompose a number into a list of characters
number_chars(-Number, @List)	Convert a list of characters into a number
number_codes(+Number, ?List)	Decompose a number into a list of character codes
number_codes(-Number, @List)	Convert a list of character codes into a number
remove(+Sublist, +List, ?Remainder)	Remove a sublist from a list
reverse(@List, ?ReversedList)	Reverse a list
sort(@List, ?SortedList)	Sort a list

Arithmetic

@Value < @Value	Value less than Value
@Value > @Value	Value greater than Value
@Value =< @Value	Value less or equal Value
@Value >= @Value	Value greater or equal Value
@Value =:= @Value	Value equal Value
@Value =\= @Value	Value not equal Value
atom_number(-Atom, @Number)	Convert a number into an atom
for(+Start, +Counter, +End)	Check range
#**for**(+Start, ?Counter, +End)	Generate a sequence of integers
?Result **is** @Expression	Evaluate arithmetic expressions

Global variables

@#**current_global**(?Name)	Query global variables
@**get_global**(+Name, ?Value)	Query the value of a global variable

@**pop_global**(+Name, ?Value)	Query global variable and remove value
@**push_global**(+Name, @Value)	Set value of a global variable
@**set_global**(+Name, @Value)	Set value of a global variable
@**unset_global**(+Name)	Delete a global variable

Stream processing

assign_alias(+Alias, @Stream)	Define an alias for an input/output stream
cancel_alias(+Alias)	Cancel an alias for an input/output stream
close(@Stream)	Close an input/output stream
close(@Stream, @Actions)	Close an input/output stream
#**current_alias**(?Alias, ?Stream)	Query aliases of input/output streams
#**current_device_control**(+Device, ?Command, ?Arity)	Query information on device drivers
current_error(?Stream)	Query current error output stream
current_input(?Stream)	Query current input stream
current_output(?Stream)	Query current output stream
#**current_stream_control**(@Stream, ?Command, ?Arity)	Query information on device drivers
device_control(+Device, @Command)	Control an input/output device
@**edit**	Edit a file
@**edit**(+Filename)	Edit a file
error_tell(@Stream)	Set current error output stream
error_telling(?Stream)	Query current error output stream
error_told	Reset current error output stream
file_test(+Pathname, +AccessMode)	Check access permission for a file
filepos(@Stream, ?Line)	Position within an input stream
filepos(@Stream, ?Line, ?Column)	Position within an input stream
flush_output	Flush the buffer of an output stream
flush_output(@Stream)	Flush the buffer of an output stream
open(@DeviceAndName, +Mode, -Stream)	Open an input/output stream
open(@DeviceAndName, +Mode, -Stream, @Options)	Open an input/output stream
reset_streams	Restore default values for current streams
see(@Stream)	Set current input stream
seeing(?Stream)	Query current input stream
seen	Reset current input stream
set_error(@Stream)	Set current error output stream

set_input(@Stream)	Set current input stream
set_output(@Stream)	Set current output stream
set_stream_position(@Stream, @Position)	Position within an input/output stream
stream_control(@Stream, @Command)	Control an input/output stream
stream_copy(@Stream1, @Stream2)	Redefine streams
stream_device(@Stream, ?Device)	Determine device name of an input/output stream
#**stream_property**(?Stream, ?Info)	Query information on input/output streams
stream_type(@Stream, ?StreamType)	Query type of an input/output stream
tell(@Stream)	Set current output stream
telling(?Stream)	Query current output stream
told	Reset current output stream
unix_make_pipe(-Pipename)	Create a pipe

Operators

#**current_op**(?Priority, ?Assoc, ?Name)	Information on operators
op(+Priority, +Assoc, @Names)	Define and delete operators
:- **op**(+Priority, +Assoc, @Names)	Define and delete operators (directive)

Elementary input/output

at_end_of_line	Query end of line
at_end_of_line(@Stream)	Query end of line
at_end_of_stream	Query end of stream
at_end_of_stream(@Stream)	Query end of stream
get_byte(?ByteCode)	Input a byte
get_byte(@Stream, ?ByteCode)	Input a byte
get_char(?Character)	Input a character
get_char(@Stream, ?Character)	Input a character
get_code(?CharCode)	Input a character
get_code(@Stream, ?CharCode)	Input a character
get_until(+SearchChar, ?Text, ?EndChar)	Read up to a specific character
get_until(@Stream, +SearchChar, ?Text, ?EndChar)	Read up to a specific character
nl	Output newline

nl(@Stream)	Output newline
outtab(+Position)	Position the cursor
outtab(@Stream, +Position)	Position the cursor
peek_byte(?ByteCode)	Input a byte
peek_byte(@Stream, ?ByteCode)	Input a byte
peek_char(?Character)	Input a character
peek_char(@Stream, ?Character)	Input a character
peek_code(?CharCode)	Input a character
peek_code(@Stream, ?CharCode)	Input a character
put_byte(+ByteCode)	Output a byte
put_byte(@Stream, +ByteCode)	Output a byte
put_char(+Char)	Output a character
put_char(@Stream, +Char)	Output a character
put_code(+CharCode)	Output a character
put_code(@Stream, +CharCode)	Output a character
skip_line	Skip an input line
skip_line(@Stream)	Skip an input line
tab(+Number)	Move the cursor forward
tab(@Stream, +Number)	Move the cursor forward

Input/output for terms

char_conversion(+CharacterIn, +CharacterOut)	Define a character conversion
:- **char_conversion**(+CharacterIn, +CharacterOut)	Define a character conversion (directive)
#**current_char_conversion**(?CharacterIn, ?CharacterOut)	Information on character conversions
#**current_op**(?Priority, ?Assoc, ?Name)	Information on operators
float_format(?FormatOld, @FormatNew)	Query and set floating-point format
op(+Priority, +Assoc, @Names)	Define and delete operators
:- **op**(+Priority, +Assoc, @Names)	Define and delete operators (directive)
portray(@Stream, @Term)	User-defined output predicate
@**print**(@Term)	Output a term
@**print**(@Stream, @Term)	Output a term
read(?Term)	Input a term
read(@Stream, ?Term)	Input a term
read_error(?Line, ?Number)	Query position of a syntax error

read_error(?Line, ?Column, ?Number)	Query position of a syntax error
read_term(?Term, +Options)	Input a term
read_term(@Stream, ?Term, +Options)	Input a term
syntax_error(+Number, ?Message)	Assignment of syntax error number to error message
write(@Term)	Output a term
write(@Stream, @Term)	Output a term
write_canonical(@Term)	Output a term
write_canonical(@Stream, @Term)	Output a term
write_formatted(+Format, @TermList)	Formatted output of terms
write_formatted(@Stream, +Format, @TermList)	Formatted output of terms
@**write_term**(@Term, @Options)	Output a term
@**write_term**(@Stream, @Term, @Options)	Output a term
writeq(@Term)	Output a term in readable form
writeq(@Stream, @Term)	Output a term in readable form

Screen control

clear_eol	Clear to end of line
clear_eol(@Stream)	Clear to end of line
clear_eos	Clear to end of screen
clear_eos(@Stream)	Clear to end of screen
clear_screen	Clear screen
clear_screen(@Stream)	Clear screen
get_last(?Line, ?Column)	Determine the cursor position
get_last(@Stream, ?Line, ?Column)	Determine the cursor position
move_cursor(+Line, +Column)	Position the cursor
move_cursor(@Stream, +Line, +Column)	Position the cursor
tty_size(?Lines, ?Columns)	Query the screen size
tty_size(@Stream, ?Lines, ?Columns)	Query the screen size

Directives

:- **begin_module**(+Module)	Define the body of a module
:- **char_conversion**(+CharacterIn, +CharacterOut)	Define a character conversion

:- **discontiguous**(@PredicateList)	Declare discontiguous predicate definition
:- **dynamic**(@PredicateList)	Declare predicates as modifiable
:- **end_module**	Conclude the definition of a module
:- **end_module**(+Module)	Conclude the definition of a module
:- **ensure_loaded**(+Filename)	Insert a Prolog file in the source text once
:- **export**(@PredicateList)	Export predicates
:- **import**(+Module)	Import predicates
:- **import**(+Module, @PredicateList)	Import predicates selectively
:- **include**(+Filename)	Include a Prolog file in the source text
:- **init**(+Goal)	Initialization goal
:- **initialization**(+Goal)	Initialization goal
:- **meta**(@PredicateList)	Declare metapredicates
:- **module**(+Module)	Define a module
:- **multifile**(@PredicateList)	Declare predicates as being scattered over several files
:- **nonotify**	Do not output load message
:- **op**(+Priority, +Assoc, @Names)	Define and delete operators
:- **private**(@PredicateList)	Declare predicates as private
:- **reexport**(+Module)	Reexport predicates
:- **reexport**(+Module, @PredicateList)	Reexport predicates
:- **set_prolog_flag**(+Flag, +Value)	Set value of a Prolog flag

Modules

+Module **:** +Predicate	Specify the module of the called predicate
+Predicate **@** +Module	Specify module context for predicate call
:- **begin_module**(+Module)	Define the body of a module
calling_context(?Module)	Query the module argument of metapredicates
compile(+Filename)	Compile a Prolog text
current_default_module(?Module)	Query current module
#**current_module**(?Module)	Query modules
:- **end_module**	Conclude the definition of a module
:- **end_module**(+Module)	Conclude the definition of a module
:- **export**(@PredicateList)	Export predicates
:- **import**(+Module)	Import predicates
:- **import**(+Module, @PredicateList)	Import predicates selectively
load(+Filename)	Load compiled Prolog text
:- **meta**(@PredicateList)	Declare metapredicates

:- **module**(+Module)	Define a module
:- **private**(@PredicateList)	Declare predicates as private
:- **reexport**(+Module)	Reexport predicates
:- **reexport**(+Module, @PredicateList)	Reexport predicates
@**save_module**(+Filename)	Store current module in a file
set_default_module(+Module)	Set current module
unload(+Module)	Remove module from the database

Grammar rules

+Arg1 `-->` +Arg2	Create a grammar rule
expand_term(+TermIn, ?TermOut)	Transform a grammar rule
#**phrase**(+Goal, ?List)	Call a grammar rule

Database manipulation

@`[` +Filename, ...`]`	List notation for consult/1
@`[ -` +Filename, ...`]`	List notation for reconsult/1
@**abolish**(@Predicate)	Remove a predicate from the database
@**asserta**(@Clause)	Insert a clause in the database
@**asserta**(@Head, @Body)	Insert a clause in the database
@**asserta_with_names**(@Clause, @VarList)	Insert a clause in the database
@**asserta_with_names**(@Head, @Body, @VarList)	Insert a clause in the database
@**assertz**(@Clause)	Insert a clause in the database
@**assertz**(@Head, @Body)	Insert a clause in the database
@**assertz_with_names**(@Clause, @VarList)	Insert a clause in the database
@**assertz_with_names**(@Head, @Body, @VarList)	Insert a clause in the database
compile(+Filename)	Compile a Prolog text
@**consult**(+Filename)	Read in a Prolog file
:- **ensure_loaded**(+Filename)	Insert a Prolog file in the source text once
:- **include**(+Filename)	Include a Prolog file in the source text
load(+Filename)	Load compiled Prolog text
load_system(+Filename)	Restore the old system status

@**reconsult**(+Filename)	Read in a Prolog file
@#**retract**(+Clause)	Remove a clause from database
@#**retract**(+Head, ?Body)	Remove a clause from database
@#**retract_with_names**(+Clause, ?VarList)	Remove a clause from database
@#**retract_with_names**(+Head, ?Body, ?VarList)	Remove a clause from database
save_runtime(+Filename)	Save system status for creating applications
save_system(+Filename)	Save current system status

Query database

@#**clause**(+Head, ?Body)	Search the database for specific clauses
@#**clause_with_names**(+Head, ?Body, ?VarList)	Search the database for specific clauses
@#**current_predicate**(?Predicate)	Query visible predicates
@#**current_visible**(?DefModule, ?Predicate)	Query visible predicates
@**listing**	List the clauses in the database
@**listing**(+Functor)	List the clauses in the database
@**listing**(@Predicate)	List the clauses in the database
@**predicate_type**(@Predicate, ?Type)	Query predicate type

Predicate attributes

@**debug_mode**(@Predicate, ?Before, +After)	Query and set the debug mode of a predicate
:- **discontiguous**(@PredicateList)	Declare discontiguous predicate definition
:- **dynamic**(@PredicateList)	Declare predicates as modifiable
:- **export**(@PredicateList)	Export predicates
:- **import**(+Module)	Import predicates
:- **import**(+Module, @PredicateList)	Import predicates selectively
@**list_mode**(@Predicate, ?Before, +After)	Query and set the list mode of a predicate
:- **meta**(@PredicateList)	Declare metapredicates
@**modify_mode**(@Predicate, ?Before, +After)	Query and set the modify mode of a predicate
:- **multifile**(@PredicateList)	Declare predicates as being scattered over several files

@**predicate_mode**(@Predicate, ?Before, +After)	Query and set the exception mode of a predicate
@**predicate_type**(@Predicate, ?Type)	Query predicate type
:- **private**(@PredicateList)	Declare predicates as private
:- **reexport**(+Module)	Reexport predicates
:- **reexport**(+Module, @PredicateList)	Reexport predicates

Execution control

!	Disable backtracking
+Goal1 **,** +Goal2	Conjunction of goals
+ConditionGoal **->** +ThenGoal	IF-THEN conjunction of goals
+ConditionGoal **->** +ThenGoal **;** +ElseGoal	IF-THEN-ELSE conjunction of goals
#+Goal1 **;** +Goal2	Disjunction of goals
\+(@Goal)	Negation through failure
abort	Return to the toplevel loop of IF/Prolog
@#**bagof**(?Term, +Goal, ?TermList)	Find selected solutions of a goal
break	Generate a new input loop
@**call**(+Goal)	Call a goal
@**call**(+Name, +ArgList)	Call a goal
fail	Initiate backtracking
@**findall**(?Term, +Goal, ?TermList)	Find all solutions of a goal
for(+Start, +Counter, +End)	Check range
#**for**(+Start, ?Counter, +End)	Generate a sequence of integers
halt	Terminate Prolog
halt(+ExitStatus)	Terminate Prolog and set exitstatus
:- **init**(+Goal)	Initialization goal
:- **initialization**(+Goal)	Initialization goal
not @Goal	Negation through failure
@**once**(+Goal)	Activate a goal once only
program	Automatically call a Prolog goal
#**repeat**	Generate a choice point
@#**setof**(?Term, +Goal, ?TermList)	Find selected solutions of a goal
true	Goal which is always true

Contexts/exceptions

alarm(+Seconds)	Schedule alarm

@**catch**(+Goal, +CatchMask, +SubstituteGoal)	Intercept a context jump
@**context**(+Goal, +ContextInfo)	Create a context
#**current_signal**(?Signal, ?Mode)	Query signals and modes
exception(@Error, @Info)	Raise an exception
@**exception_handler**(+Goal, +Exception, +Handler)	Define local exception handling
signal_control(+Signal, +Command)	Control signal processing
@**signal_handler**(+Goal, ?Signal, +Handler)	Define local signal handling
throw(@Ball)	Initiate a context jump

Debugger

@**debug**(+Goal)	Activate the debugger for a goal
debug_begin	User-defined debugger initialization
debug_config(+Parameter, ?Old, +New)	Configure the debugger
debug_end	User-defined termination of the debugger
@**debug_mode**(@Predicate, ?Before, +After)	Query and set the debug mode of a predicate

Tracer

@**profile**(+Goal)	Activate profiling for a goal
@**profile**(+Goal, @Ports)	Activate profiling for a goal
profile_reset	Reset profiling statistics
profile_stat	Query profiling statistics
profile_stat(?List)	Query profiling statistics
@**trace**(+Goal)	Activate tracing for a goal
trace_begin	User-defined tracer initialization
trace_config(+Parameter, ?Old, +New)	Configure the tracer
trace_end	User-defined termination of the tracer
trace_output(@CallNumber, +Port, @Goal, @VariableNames)	User-defined output predicate for the tracer

Memory management

#**current_memory_-management**(?Area, ?Parameter, ?Value)	Query memory management parameters
garbage_collection(+Area)	Activate garbage collection
set_memory_management(+Area, +Parameter, +Value)	Set memory management parameters
statistics	Print system values
statistics(?Type, ?ResultList)	Query system values

System information

#**current_prolog_flag**(?Flag, ?Value)	Query Prolog flags
help	Display information about help
help(+Pattern)	Display help information
manual	Output manual pages
manual(+Functor)	Output manual pages
manual(@Predicate)	Output manual pages
program_parameters(?ParameterList)	Query parameters from IF/Prolog call
prolog_serial(?Serial)	Query IF/Prolog serial number
prolog_version(?Version)	Query IF/Prolog version
proroot(?Path)	Query path of IF/Prolog system
set_prolog_flag(+Flag, +Value)	Set value of a Prolog flag
:- **set_prolog_flag**(+Flag, +Value)	Set value of a Prolog flag (directive)
system_hostid(?Hostid)	Query machine identification
system_name(?SystemName)	Query operating system
system_parameters(?ParameterList)	Query system parameters from IF/Prolog call
user_parameters(?ParameterList)	Query user parameters from IF/Prolog call

Operating system environment

chdir(+Dir)	Change current directory
default_editor(?EditorOld, +EditorNew)	Query and set the editor
@**edit**	Edit a file
@**edit**(+Filename)	Edit a file
exec(+Command)	Start program execution
exec(+Prog, @Args)	

file_test(+Pathname, +AccessMode)	Check access permission for a file
#**get_file_info**(+Pathname, ?Attribute, ?Value)	Get OS information for file
get_file_info(+Pathname, ?Info)	Get OS information for file
getcwd(?Dir)	Query current working directory
getenv(+Name, ?Value)	Query environment variable
localtime(+Time, ?Year, ?Month, ?Day, ?DoW, ?DoY, ?Hour, ?Min, ?Sec)	Determine the date and time
stream_copy(@Stream1, @Stream2)	Redefine streams
system	Start the operating system command interpreter
system(+Command)	Execute an operating system command
system(+Command, ?ExitStatus)	Execute an operating system command
system(+Command, ?Input, ?Output)	Execute an operating system command with input and output specified
system(+Command, ?Input, ?Output, ?Error, ?Pid)	Execute an operating system command with input and output specified
timezone (+Time, ?TimeZone, ?AlternateZone, ?DaylightSaving)	
unix_fork(?Pid)	Spawn process
unix_getpid(?Pid)	Query process ID of current process
unix_kill(+Pid, +Signal)	Send signal to a process
unix_make_pipe(-Pipename)	Create a pipe
unix_wait(?Pid)	Wait for termination of a child process
unix_wait(?Pid, ?ExitStatus)	Wait for termination of a child process
windows_chdrive(+Drivename)	Change drive
windows_getdrive(?Drivename)	Query drive

Processes

exec(+Command)	Start program execution
exec(+Prog, @Args)	Start program execution
system	Start the operating system command interpreter
system(+Command)	Execute an operating system command
system(+Command, ?ExitStatus)	Execute an operating system command
system(+Command, ?Input, ?Output)	Execute an operating system command with input and output specified
system(+Command, ?Input, ?Output, ?Error, ?Pid)	Execute an operating system command with input and output specified

unix_fork(?Pid)	Spawn process
unix_getpid(?Pid)	Query process ID of current process
unix_kill(+Pid, +Signal)	Send signal to a process
unix_wait(?Pid)	Wait for termination of a child process
unix_wait(?Pid, ?ExitStatus)	Wait for termination of a child process

Net communication

#**current_host**(?Host)	Query names of current host
#**current_socket**(?Domain, ?Type, ?Socket)	Query communication sockets
#**get_socket_option**(@Socket, ?Option, ?Value)	Get socket option
#**host_addr**(+Host, ?Addr)	Query host address
#**host_addr**(−Host, +Addr)	Query host name
net_service(+Service, ?Protocol, ?Port)	Query communication service
select(@Streams, +Timeout, −ReadyReadStreams)	Synchronous input multiplexing
select(@Streams, +Timeout, −ReadyReadStreams, −ReadyWriteStreams)	Synchronous input multiplexing
select(@SocksOrStreams, +Timeout, −ReadyReadSOS, −ReadyWriteSOS, −Timeleft)	Synchronous input multiplexing
set_socket_option(@Socket, +Option, +Value)	Set socket option
socket(+Domain, +Type, −Socket)	Create communication socket
socket_accept(@Socket, ?Addr, −NewSocket)	Accept a connection
socket_bind(@Socket, ?Addr)	Bind a name to a socket
socket_close(@Socket)	Close a socket
socket_connect(@Socket, @Addr)	Connect a socket
socket_listen(@Socket)	Listen for connections
socket_listen(@Socket, +Quelen)	Listen for connections
socket_raw_receive(@Socket, @Size, ?List, ?Length)	Receive data from socket
socket_raw_receive(@Socket, @Size, ?Addr, ?List, ?Length)	Receive data from socket
socket_receive(@Socket, ?List)	Receive data from socket

socket_receive(@Socket, ?Addr, ?List)	Receive data from socket
socket_send(@Socket, @List)	Send data to socket
socket_send(@Socket, @Addr, @List)	Send data to socket
socket_shutdown(@Socket)	Close a socket
socket_shutdown(@Socket, +How)	Close a socket

Chapter 4

Reference section for the predefined predicates

This section contains descriptions of the built-in predicates in alphabetical order. Predicates which are adjacent to each other in the alphabet and which are similar in function are described together.

The language constructs, directives and the special user-defined predicates used by IF/Prolog can also be found here.

Predicates which permit backtracking are indicated by a hash character ('`#`') in front of the functor. This character does not appear with predicates not having this capability.

For **metapredicates**, the argument list is followed by the text `[` @ `+`*Module* `]`. The module qualification can be specified by `@/2`. **Metapredicates** are supplied with information on the calling module.

Directives are special syntactic structures which can be specified in IF/Prolog texts and which are processed when these texts are read in (e.g. with `consult/1`). They are indicated by `:-` in front of the functor, as they would be written in a Prolog text.

User definable predicates are automatically called at particular points by IF/Prolog if the user has defined them. They are identified by the text `[ :-` *Body* `]` after the argument list.

The **call mode** specifies the instantiation of an argument at the time of the call. In front of each argument there is a sign ('`@`', '`+`', '`-`', or '`?`') to indicate the call type of the argument, as follows:

@ The argument is a pure **input** parameter. The current parameter specified in the call must be of the prescribed type and any uninstantiated variables contained in this parameter are not instantiated in the call.

+ The argument is an **input** parameter. The current parameter specified in the call must be of the prescribed type. Any uninstantiated variables contained in this parameter may be instantiated in the call.

When the argument is an atomic term, there is no difference between the modes + and @. The mode @ is therefore used only when the argument may be a compound term.

? The argument is an **input/output** parameter. The current parameter must be either a variable or a term of the prescribed type. In the course of the execution of the predicate, this parameter is unified. If this unification is not successful, then the entire predicate call will fail. Any uninstantiated variables contained in this parameter may be instantiated in the call.

- The argument is a pure **output** parameter. The current parameter must be an uninstantiated variable. If the predicate succeeds, this variable is instantiated with the result of the predicate call. The type of result from the predicate call is indicated in the section on '*Arguments*' in the full description.

Several call patterns are possible for some predicates.

Disable backtracking

!

The execution of `!/0` (**cut**) succeeds once and once only. If the **cut** is called as a subgoal when executing a goal G, all the choice points set when executing G are destroyed.

This has the following effects:

- While executing G, Prolog is now committed to the clause where the cut was called as a subgoal.
- All instantiations made till now when executing G become permanent.

If the cut is again reached from the right in the course of backtracking, the execution of the calling goal G will fail, because it is impossible to make any new variable instantiations or to select new clauses.

A further interpretation of the cut is that a goal can be executed precisely once and in precisely one way if the cut is called at a suitable position (normally at the start of the body of a clause). Backtracking is not performed.

The cut has an important function in loops which are constructed using the predicate `repeat/0`. When used there, it prevents a loop from being reentered during backtracking once the termination condition has been encountered.

Hints

`!/0` greatly reduces the readability of programs. The cut should therefore be used very sparingly. The use of language construct `->/2` can avoid many traditional applications of `!/0`.

The language construct `!/0` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

Example

In contrast to `append/3`, the predicate `append_x/3` defined here searches only for the first possible solution and ignores any alternatives.

```
[user] ?- listing.<ENTER>

% *** user: append_x / 3 ***
```

```
append_x([],L,L) :-
    ! .
append_x([H|T],L,[H|EL]) :-
    append_x(T,L,EL) .

yes

[user] ?- append_x(X,Y,[a,b,c]).<ENTER>

X       = []
Y       = [a,b,c] ;<ENTER>

no
```

In order to state that Indiana Jones likes all animals except snakes, you could say that the relationship `indiana_jones likes snake` is to be evaluated with fail. If the cut were to be omitted from the following program, the second clause for `likes/2` would be found after backtracking and the goal `likes(indiana_jones, snake)` could be executed. The cut prevents backtracking and specifies that `likes(indiana_jones, snake)` can be executed either exactly once or, as in this case due to fail, not at all.

```
[user] ?- listing.<ENTER>

% *** user: animal / 1 ***
animal(snake).
animal(tiger).

% *** user: likes / 2 ***
likes(indiana_jones,snake) :-
    !,
    fail .
likes(indiana_jones,X) :-
    animal(X) .

yes

[user] ?- likes(indiana_jones,snake).<ENTER>

no

[user] ?- likes(indiana_jones,tiger).<ENTER>

yes
```

However, this definition has its disadvantages. If you ask whether `indiana_jones` likes anything, you receive an answer which conflicts with that given above.

```
[user] ?- likes(indiana_jones,X).<ENTER>

no
```

The cut which is normally called for the predicate `likes` prevents the instantiation of X to `snake` from being undone, and thus prevents a search for alternatives (in this case `tiger`).

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

repeat/0, fail/0, true/0

Conjunction of goals

+*Goal1* , +*Goal2*
','(+*Goal1*, +*Goal2*)

The language construct `,/2` is used to combine two goals to form a composite goal.

The new goal succeeds if both *Goal1* and *Goal2* succeed. *Goal2* will be executed only if *Goal1* succeeds.

If *Goal1* or *Goal2* are re-executable, they can return several results. Backtracking takes place first in *Goal2* and, if that is not (or no longer) possible, in *Goal1*. *Goal2* will then be re-executed, if *Goal1* succeeds.

Arguments

Goal1	Goal
Goal2	Goal

Exceptions

instantiation_error
The argument *Goal1* or *Goal2* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal1* or *Goal2* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal1* or *Goal2*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal1* or *Goal2* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal1* or *Goal2* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The arguments *Goal1* and *Goal2* may themselves be composite goals.

The atom `','` is defined as an infix operator.

The language construct `,/2` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs.

Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

The predicates activated in the goal must be visible in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of the specified *Module*, unless the `@/2` qualification is used for a meta-predicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- member(X,[a,b,c]),member(X,[b,d,a]).<ENTER>

X       = a ;<ENTER>

X       = b ;<ENTER>

no

[user] ?- X is 7,<ENTER>
          X < 10, X > 3.<ENTER>

X       = 7 <ENTER>

yes

[user] ?- \+ fail, true.<ENTER>

yes
```

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

;/2, call/1, @/2, :/2, set_prolog_flag/2

Create a grammar rule

+*Arg1* --> +*Arg2*
'-->'(+*Arg1*, +*Arg2*)

The atom `-->` does not denote a built-in predicate but is a built-in operator which is interpreted in a special way by `consult/1`, `reconsult/1` and `compile/1`.

Clauses formulated as shown above are converted into *normal* Prolog rules when they are read in with `consult/1`, `reconsult/1` or `compile/1`. These Prolog rules are normally used for parsing lists of terms (characters in mosts cases).

This mechanism can be used to parse natural language or to define sentences of a formal language.

The operator notation with `-->` illustrates the constituent structure. The higher-level constituent, which is always non-terminal, is on the left of the arrow. One or more terminal and/or non-terminal constituents are specified to the right of the arrow.

The conversion mechanism for `-->` ensures that both *Arg1* and *Arg2* are supplied with additional arguments.

A method called the differential list method is used. The first argument added by the mechanism is a list which starts with the string of terms to be parsed. The second argument is a list containing the remainder of the first list (without the first part, which is to be parsed).

Arguments

Arg1	Atom, structure
Arg2	Atom, structure

Hints

If you specify additional arguments in *Arg1* or *Arg2*, then these are inserted **before** the arguments generated by the system during conversion into Prolog clauses.

A detailed description of the Prolog grammar rules can be found in *IF/Prolog User's Guide*[2].

Example

An English sentence can, very roughly, be described as a sequence consisting of a nominal phrase and a verbal phrase. In arrow notation, it would look like this:

```
sentence --> nom_phrase, verb_phrase.
nom_phrase --> [dogs].
verb_phrase --> [bark].
```

As soon as these clauses have been read into your database with `consult/1` or `reconsult/1`, you can have a look at their structure by using `listing/0/1` for example:

```
[user] ?- listing.<ENTER>

% *** user: sentence / 2 ***
sentence(A,B) :-
    nom_phrase(A,C),
    verb_phrase(C,B) .

% *** user: nom_phrase / 2 ***
nom_phrase([dogs|A],A).

% *** user: verb_phrase / 2 ***
verb_phrase([bark|A],A).

yes
```

The rules can be stated like this: A list *L*, minus a list *L0*, is a sentence if there is a list *L1* such that the first part of *L* without *L1* is a nominal phrase and *L1* itself, minus the list *L0*, is a verbal phrase.

A list minus a list *L2* is a nominal phrase if the difference consists precisely of the atom *dogs*.

A list minus a list *L3* is a verbal phrase if the difference consists precisely of the atom *bark*. This grammar is correct, although it is by no means complete. However, you can see from it how to parse sentences.

```
[user] ?- sentence([dogs,bark],[]).<ENTER>

yes
```

As you can see, the mechanism can also be used to generate sentences, even though this grammar is so small that it contains only one sentence.

```
[user] ?- sentence(X,[]).<ENTER>

X       = [dogs,bark] ;<ENTER>

no
```

Standard

This language construct is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

expand_term/2, phrase/2

IF-THEN conjunction of goals

+*ConditionGoal* -> +*ThenGoal* '->'(+*ConditionGoal*, +*ThenGoal*)

+*ConditionGoal* -> +*ThenGoal* ; +*ElseGoal* ';'('->'(+*ConditionGoal*, +*ThenGoal*), +*ElseGoal*)

The language construct `->/2` is used to combine two goals to form a new goal.

The new goal succeeds if both *ConditionGoal* and *ThenGoal* succeed. *ThenGoal* will be executed only if *ConditionGoal* succeeds.

In contrast to `,/2`, *ConditionGoal* is executed in a special way:

- Backtracking by *ConditionGoal* is not possible. The composite goal can therefore only return several results if *ConditionGoal* succeeds and *ThenGoal* returns several results.
- If *ConditionGoal* itself is a composite goal and if it contains a **cut** (`!/0`), then this cut will have no effect beyond the *ConditionGoal*.

If the new goal produced with `->/2` occurs as the first subgoal in a disjunction (OR construct, `;/2`) then `;/2` behaves differently. This is similar to "if-then-else" constructs in other programming languages.

The new goal produced with `->/2` and `;/2` succeeds if either *ConditionGoal* and *ThenGoal* succeed or *ConditionGoal* fails and *ElseGoal* succeeds.

In procedural view, this means that *ConditionGoal* is first executed. If this is successful, *ThenGoal* is executed and the composite goal succeeds if *ThenGoal* succeeds; otherwise it fails. If *ConditionGoal* fails, *ElseGoal* is executed and the composite goal succeeds if *ElseGoal* succeeds; otherwise it fails.

The subgoal *ThenGoal* is therefore only executed if *ConditionGoal* succeeds and *ElseGoal* is only executed if it fails.

By backtracking, *ThenGoal* and *ElseGoal* can have several solutions.

Arguments

ConditionGoal	Goal
ThenGoal	Goal
ElseGoal	Goal

Exceptions

instantiation_error
The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *ConditionGoal*, *ThenGoal* or *ElseGoal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

Using this language construct enables you to avoid many applications of cut (`!/0`).

The call

```
ConditionGoal -> ThenGoal
```

is equivalent to

```
ConditionGoal -> ThenGoal ; fail .
```

The atom `'->'` is defined as an infix operator.

The language construct `->/2` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

The predicates activated in the goal must be visible in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

Using the IF-THEN construct, the predicate `append_x/3` (see `!/0`) can be defined with only one clause and without a cut:

```
[user] ?- listing.<ENTER>

% *** user: append_x / 3 ***
append_x(L1,L2,L3) :-
        L1 = []
    ->
        L2 = L3
    ;
        L1 = [X|T1],
        L3 = [X|T3],
        append_x(T1,L2,T3) .

yes

[user] ?- append_x([a],[b,c],L).<ENTER>

L       = [a,b,c] <ENTER>

yes

[user] ?- append_x(X,Y,[a,b]).<ENTER>

X       = []
Y       = [a,b] ;<ENTER>

no
```

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

;/2, !/0, if/3, call/1, @/2, :/2, set_prolog_flag/2

List notation for consult/1, reconsult/1

```
[ +Filename, ...] [ @ +Module ]
[ - +Filename, ...] [ @ +Module ]
'.'( +Filename, + Filelist ) [ @ +Module ]
'.'( '-'( +Filename ), + Filelist ) [ @ +Module ]
```

The predicate `./2` is functionally identical to `consult/1` and `reconsult/1`. It is simply an abbreviated notation, particularly if a number of files are to be consulted or reconsulted.

If an element of the list has the form "`-` *Filename*", then `reconsult/1` is called for this *Filename*, otherwise `consult/1`. Elements are processed one by one.

Arguments

Filename	Atom, name of a file
Filelist	List of atoms or structures with the form `-`*Filename*

Exceptions

type_error(list)
The argument *Filelist* must be a regular list, but is a term of another type or not regular.

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Filename*.

In addition, all the error messages and warnings from `read_term/2`, `assertz/1` and `assertz_with_names/2` and may occur. These do not cause the load process to be aborted. You should be aware that the database may not necessarily contain what you expect it to contain.

Errors in directives are also output as error messages on the error output stream. Unknown directives are reported by the error message `domain_error(directive)`.

In the event of errors in a module interface, the affected module is deleted after *Filename* has been processed.

If there are errors in the declaration part of a module body, the affected module body is deleted after *Filename* has been processed.

When initialization directives are executed (see `init/1`), any exception may arise. These aborts the execution of the initialization goal, but does not abort the load process.

In addition, warnings are output as a function of the Prolog flag `warnings` in certain situations to alert you to possible sources of errors.

Hints

If *Filename* does not contain the extension *.pro* and if a file of this name cannot be found, the extension *.pro* is appended and the resultant filename used.

The predicate `./2` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

If *Filename* contains module definitions, then database manipulation takes place in these modules. If not, it is implicitly assumed that the contents of *Filename* belong to the body of the calling module or of the specified *Module*. The difference compared with an explicitly specified `body/1` directive at the start of *Filename* is that the associated body of the module is not first deleted.

If *Filename* contains only module interfaces and/or bodies, then `consult/1` and `reconsult/1` are prefectly identical. The definition of a module interface of an existing module with the same name is first deleted, and the definition of a module body overwrites an existing module body.

The `init/1` directives in *Filename* are also executed in the context of the relevant module.

If a module in *Filename* contains several `init/1` directives, then the execution sequence is undefined.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

consult/1, reconsult/1

Specify the module of the called predicate

+*Module* **:** +*Predicate*
':'(+*Module*, +*Predicate* **)**

This notation indicates that execution of *Predicate* is to be based on its visibility in *Module*.

It is not necessary for *Predicate* to have been exported from this module or for *Module* to have been imported in the current module. The definition of the same predicate in the current module or in another imported module is ignored.

Predicate must not be a language construct or a qualified predicate.

Arguments

Module	Atom, name of a module
Predicate	Atom or structure

Exceptions

The possible errors relating to this notation are detected and reported by IF/Prolog during interpretation of a goal (`call/1`) or on insertion in the database (`asserta/1/2`, etc.).

Hints

The atom `':'` is defined as an infix operator.

A predicate call cannot be qualified more than once with `:`.

Standard

This language construct is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

@/2, call/1, module/1, import/1/2, export/1, reexport/1/2

Disjunction of goals

```
# +Goal1 ; +Goal2
# ';'( +Goal1, +Goal2 )
```

The language construct `;/2` is used to combine two goals to form a composite goal.

The new goal succeeds if *Goal1* succeeds or, if it does not (or does no longer) succeed, if *Goal2* succeeds. The subgoal *Goal2* is therefore only executed if *Goal1* does not (or does no longer) succeed.

If *Goal1* or *Goal2* are re-executable, they can return several results. Backtracking takes place first in *Goal1* and, if that is not (or no longer) possible, in *Goal2*.

If subgoal *Goal1* is a composite goal produced with `->/2`, then `;/2` has a different effect from the one described above. This combination resembles the "if-then-else" constructs of other programming languages, and is described under `->/2`.

Arguments

Goal1	Goal
Goal2	Goal

Exceptions

instantiation_error
The argument *Goal1* or *Goal2* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal1* or *Goal2* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal1* or *Goal2*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal1* or *Goal2* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal1* or *Goal2* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The arguments *Goal1* and *Goal2* may themselves be composite goals.

The atom `';'` is defined as an infix operator.

The language construct `;/2` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

The predicates activated in the goal must be visible in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- member(X,[a,b]);member(X,[c]).<ENTER>

X       = a ;<ENTER>

X       = b ;<ENTER>

X       = c ;<ENTER>

no

[user] ?- member(b,[a]);member(b,[b]).<ENTER>

yes
```

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

->/2, , /2, call/1, @/2, :/2, set_prolog_flag/2

Compare arithmetic expressions

@*Value1* < @*Value2* '<'(@*Value1*, @*Value2*)

@*Value1* > @*Value2* '>'(@*Value1*, @*Value2*)

@*Value1* =< @*Value2* '=<'(@*Value1*, @*Value2*)

@*Value1* >= @*Value2* '>='(@*Value1*, @*Value2*)

@*Value1* =:= @*Value2* '=:='(@*Value1*, @*Value2*)

@*Value1* =\= @*Value2* '=\='(@*Value1*, @*Value2*)

The comparison predicates are used to compare numeric values. A numeric value is a number or an arithmetic expression. Arithmetic expressions are evaluated before the comparison is performed. The call succeeds if *Value1* has the specified relationship to *Value2*.

The names of these predicates are defined as operators with associativity `xfx` and precedence 700. The table below provides an overview of the function of the predicates.

Predicate	Function
`<`	succeeds if *Value1* is less than *Value2*
`>`	succeeds if *Value1* is greater than *Value2*
`=<`	succeeds if *Value1* is less than or equal to *Value2*
`>=`	succeeds if *Value1* is greater than or equal to *Value2*
`=:=`	succeeds if *Value1* is equal to *Value2*
`=\=`	succeeds if *Value1* is not equal to *Value2*

Arguments

Value1	Number or arithmetic expression
Value2	Number or arithmetic expression

Exceptions

instantiation_error
The argument *Value1* or *Value2* or a subterm must not be a variable, but a variable was specified.

Furthermore, if an arithmetic expression is specified for the argument *Value1* or *Value2*, all the exceptions for `is/2` can occur.

Hints

Due to the limited precision when calculating with floating-point numbers, a comparison of arithmetic expressions does not always yield the mathematically correct result. Since different computers have different levels of precision, the same comparison of arithmetic expressions, when performed on different computers, may lead to conflicting results, so it is best to avoid checking floating-point numbers for precise equality.

Example

Testing two numbers for "less than" relationship:

```
[user] ?- X = 12, Y = 17, X < Y .<ENTER>

X       = 12
Y       = 17 <ENTER>

yes
```

Testing two numbers for "greater than" relationship:

```
[user] ?- X = 12, Y = 15, X > Y .<ENTER>

no
```

Testing two numbers for "less than or equal to" relationship:

```
[user] ?- X = 20, Y = 12, X =< Y .<ENTER>

no
```

Testing two numbers for "greater than or equal to" relationship:

```
[user] ?- X = 12, Y = 12, X >= Y .<ENTER>

X       = 12
Y       = 12 <ENTER>

yes
```

Testing two numbers for "equal to" relationship:

```
[user] ?- X = 8, Y = 8.0000, X =:= Y .<ENTER>

X       = 8
Y       = 8 <ENTER>

yes
```

Testing two numbers for "not equal to" relationship:

```
[user] ?- X = 12, Y = 10, X =\= Y .<ENTER>

X       = 12
Y       = 10 <ENTER>

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

The set of permissible arithmetic expressions is an extension of the set of expressions prescribed by the ISO standard (see `is/2`).

See also

is/2, op/3

Unify terms

?*Term1* = ?*Term2* =(?*Term1*, ?*Term2*)

The predicate `=/2` unifies *Term1* with *Term2*.

If neither term contains a variable, then `=/2` is equivalent to the term comparison `==/2`.

For normal unification, the Prolog system does not perform an **occurs check**; in other words, the system does not check whether one of the terms to be unified contains the other as a subterm. Any attempt to unify such a term will either result in a endless loop in the Prolog system or produce cyclical structures. Cyclical terms are output with a special notation.

Endless loops in the Prolog system do not necessarily result in an exception (e.g. with `resource_error`) and controlled exit since **garbage collection** and dynamic stack adjustment mean that memory bottlenecks are not inevitable. It is therefore possible for program crashes to occur which are based in the C runtime system and which cannot be prevented by Prolog (e.g. C runtime stack overflows).

Arguments

Term1	Term
Term2	Term

Hints

The atom `=` is built in as an infix operator.

Example

The following example constructs a cyclic list which normally leads to endless loops on output. In our example this is prevented by limiting the structure depth for output and by undoing the instantiation by means of the final `fail/0`:

```
[user] ?- [ a , R ] = R,<ENTER>
          write_term(R,[maxdepth(10)]),nl,<ENTER>
          fail.<ENTER>
[a,[a,[a,[a,[a,[a,[a,[a,[a,[a,[a,...]]]]]]]]]]]

no
```

The following examples demonstrate the normal behavior of `=/2`:

```
[user] ?- X = a * b.<ENTER>

X       = a * b <ENTER>

yes

[user] ?- f(X,Y) = f(3,4).<ENTER>

X       = 3
Y       = 4 <ENTER>

yes

[user] ?- [ A, 3 ] = [ 7 | B ].<ENTER>

A       = 7
B       = [3] <ENTER>

yes

[user] ?- a + b = +(A,B).<ENTER>

A       = a
B       = b <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

unify_with_occurs_check/2, =../2, ==/2, =:=/2, @=/2, cyclic/1

Convert between a structure and a list

+*Structure* =.. ?*List*
-*Structure* =.. +*List*
=..(+*Structure*, ?*List*)
=..(-*Structure*, +*List*)

The predicate `=../2` converts a structure into a list or a list into a structure.

A distinction can be made between two cases:

1. *Structure* is instantiated
   In this case, *List* must be a variable or a list. A list is formed from *Structure* in which the first element is the functor of *Structure* and subsequent elements are the arguments of *Structure*. The resultant list is unified with *List*. If *Structure* is a constant, then the constructed list contains only this constant.

2. *Structure* is uninstantiated
   In this case, *List* must have at least one element. If *List* contains precisely one element, this must be a constant and *Structure* is unified with this element.

   If *List* consists of more than one element, the first element must be an atom and a structure is constructed from *List*. The structure has this first element as the functor and the remaining elements of *List* as arguments. *Structure* is then unified with this structure.

Arguments

Structure	Structure, constant
List	List (regular and not empty)

Exceptions

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

instantiation_error
The first element of *List* must not be a variable, but a variable was specified.

type_error(atom)
List is a list whose head is neither an atom nor a variable, and whose tail is not the empty list.

type_error(atomic)
List is a list whose head is a compound term, and whose tail is the empty list.

domain_error(non_empty_list)
The argument *List* must be a non empty list, but is empty.

representation_error(max_arity)
The list length of the argument *List* must be less than or equal to 127, but is larger.

Hints

The atom `=..` is built in as an infix operator.

Example

Each structure can be converted into a list:

```
[user] ?- f(a,b,g(c)) =.. L.<ENTER>

L       = [f,a,b,g(c)] <ENTER>

yes

[user] ?- [a,b] =.. L.<ENTER>

L       = ['.',a,[b]] <ENTER>

yes

[user] ?- f(X) =.. [f,1].<ENTER>

X       = 1 <ENTER>

yes
```

Structures can be generated in order to form goals dynamically:

```
[user] ?- read(Func), read(Arg),<ENTER>
          Goal =.. [Func,Arg],<ENTER>
          call(Goal).<ENTER>
atom.<ENTER>
abcd.<ENTER>

Func    = atom
Arg     = abcd
Goal    = atom(abcd) <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

=/2, \=/2, atomic/1, unify_with_occurs_check/2

Compare terms for identity

@*Term1* == @*Term2* ==(@*Term1*, @*Term2*)

@*Term1* \== @*Term2* \==(@*Term1*, @*Term2*)

The predicate `==/2` succeeds if *Term1* and *Term2* are identical, otherwise it fails.

The predicate `\==/2` succeeds if *Term1* and *Term2* are not identical, otherwise it fails.

Unification does not take place, so different variables are not identical.

Arguments

Term1	Term
Term2	Term

Hints

The atoms `==` and `\==` are built in as infix operators.

Example

Different variables cannot be identical:

```
[user] ?- X == Y.<ENTER>

no

[user] ?- _ \== _.<ENTER>

yes

[user] ?- f(X) == f(Y).<ENTER>

no
```

If a variable is instantiated to a term, it is identical to that term:

```
[user] ?- X = Y, X == Y.<ENTER>

X       = _68
```

```
Y       = _68 <ENTER>

yes

[user] ?- X = 4, Y = 17 + X,<ENTER>
          Z = 17 + 4, Y == Z.<ENTER>

X       = 4
Y       = 17 + 4
Z       = 17 + 4 <ENTER>

yes
```

The next predicate `not_in(Term,List)` checks whether a term has not yet occurred in the list.

```
[user] ?- listing.<ENTER>

% *** user: not_in / 2 ***
not_in(_,[]).
not_in(T,[Head|Tail]) :-
    Head \== T,
    not_in(T,Tail) .

yes

[user] ?- not_in(3,[a,b,c]).<ENTER>

yes

[user] ?- not_in(A,[a,b,Z]).<ENTER>

A       = _68
Z       = _69 <ENTER>

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

See also

=/2, \=/2, @</2, @=</2, @>/2, @>=/2

Compare terms

@*Term1* **@<** **@***Term2*
@<(**@***Term1*, **@***Term2* **)**

@*Term1* **@>** **@***Term2*
@>(**@***Term1*, **@***Term2* **)**

@*Term1* **@=<** **@***Term2*
@=<(**@***Term1*, **@***Term2* **)**

@*Term1* **@>=** **@***Term2*
@>=(**@***Term1*, **@***Term2* **)**

@*Term1* **@=** **@***Term2*
@=(**@***Term1*, **@***Term2* **)**

@*Term1* **@\=** **@***Term2*
@\=(**@***Term1*, **@***Term2* **)**

The term comparison predicates check the relationship between *Term1* and *Term2*. The following relationships apply to terms of different types:

Variable	@<	Floating point
Floating point	@<	Rational number
Rational number	@<	Integer
Integer	@<	Atom
Atom	@<	Structure

The following applies to terms of the same type:

Variable The order is undefined and the result is therefore random.

Floating point Arithmetic order.

Rational number Arithmetic order.

Integer Arithmetic order.

Atom Character code order.

Structure Ordered primarily according to their arity, then by the name of the functor, and finally by the arguments from left to right.

Arguments

Term1	Term
Term2	Term

Hints

The atoms `@<`, `@=<`, `@>`, `@>=`, `@=` and `@\=` are built in as infix operators.

Although two variables will always be ordered at any one time, there is no guarantee that at some later time a comparison of the same two variables will yield the same result. For this reason, you should never make references to the order of variables.

Example

```
[user] ?- adam @< zeppelin.<ENTER>

yes

[user] ?- 3.77 @=< 1.<ENTER>

yes

[user] ?- 13 @> hello.<ENTER>

no

[user] ?- a(x) @>= b(c).<ENTER>

no

[user] ?- f(x) @=< f(y).<ENTER>

yes

[user] ?- g(z) @=< g(x,y).<ENTER>

yes

[user] ?- f(x) @=< a(x,y).<ENTER>

yes
```

Standard

The predicates `@<`, `@>`, `@=<` and `@>=` comply with the definition of the ISO standard for Prolog.

See also

compare/3, ==/2, \==/2

Test for non-unifiability

@*Term1* \= @*Term2*

The predicate `\=/2` succeeds, if *Term1* and *Term2* are not unifiable, otherwise it fails.

Arguments

Term1	Term
Term2	Term

Hints

The atom `\=` is built in as an infix operator.

The predicate is semantically identical to `\+(` *Term1* = *Term2*), but is not necessarily so defined!

There is no instantiation of variables *Term1* or *Term2*.

Example

```
[user] ?- X = 47 * 11,<ENTER>
          Y is 47 * 11,<ENTER>
          X \= Y.<ENTER>

X       = 47 * 11
Y       = 517 <ENTER>

yes

[user] ?- a(x,y) \= a(z,Y).<ENTER>

Y       = _71 <ENTER>

yes

[user] ?- [a,b] \= X.<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

=/2, unify_with_occurs_check/2, \+/1, not/1

Specify module context for predicate call

+*Predicate* @ +*Module*
'@'(+*Predicate*, +*Module*)

This qualification specifies that *Predicate* is to be called in the context of the module *Module*, i.e. with *Module* as the current module.

This is meaningful only for metapredicates. The qualification is ignored for other predicates because it does not matter in which context they are called.

If a metapredicate is called without this qualification, the call takes place in the context of the current module.

The specified predicate must be visible in the current module unless the defining module for *Predicate* is specified using `:` as a qualification.

Predicate must not be a language construct other than `call/1/2`.

Arguments

Predicate	Atom or structure
Module	Atom, name of a module

Exceptions

The possible errors relating to this notation are detected and reported by IF/Prolog during interpretation of a goal (`call/1`) or on inclusion in the database (`asserta/1/2`, etc.).

Hints

The atom `'@'` is defined as an infix operator.

A predicate call cannot be qualified more than once with `@`.

Standard

This language construct is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

:/2, module/1, import/1/2, export/1, reexport/1, meta/1

Negation through failure

\+ @*Goal* \+(@*Goal*)

The language construct `\+/1` calls *Goal* and fails if *Goal* succeeds. Otherwise, `\+/1` succeeds.

Arguments

Goal	Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The atom `\+` is defined as a prefix operator.

The language construct `\+/1` is identical to `not/1`.

The language construct `\+/1` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

The predicates activated in the goal must be visible in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed

in the context of the specified *Module*, unless the `@/2` qualification is used for a meta-predicate to indicate explicitly the module context in which this predicate is to be executed.

Example

The language construct `\+/1` can be used to call a goal without variables being instantiated:

```
[user] ?- \+(\+(member(X,[a,b]))).<ENTER>

X       = _68 <ENTER>

yes
```

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

not/1, call/1, fail/0, @/2, :/2, set_prolog_flag/2

Remove a predicate from the database

abolish(@*Predicate* **)** `[` **@** `+`*Module* `]`

The predicate `abolish/1` removes all clauses matching the predicate indicator *Predicate* from the database, based on its visibility in the calling module or in the specified *Module*.

Attributes of the predicate defined by the declarations `dynamic/1` or `meta/1` are unaffected.

If *Predicate* does not exist in the calling module or in the specified *Module*, `abolish/1` still succeeds.

Arguments

Predicate	Structure of the form *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
: The argument *Predicate* or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
: The argument *Predicate* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
: The name of the predicate *Predicate* must be an atom.

type_error(integer)
: The arity of the predicate *Predicate* must be an integer.

domain_error(not_less_then_zero)
: The arity of the predicate *Predicate* must be a positive integer, but is negative.

representation_error(max_arity)
: The arity of the predicate *Predicate* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
: The predicate indicator *Predicate* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

permission_error(modify,static_procedure)
: The argument *Predicate* specifies an existing predicate which has not been declared as dynamic and therefore cannot be modified.

Hints

The predicate `abolish/1` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

A predicate may have certain attributes, which, when it is created, are set implicitly by the system according to the current environment or declarations. They can be modified to a certain extent explicitly by the user. These include the following:

- List mode (see `list_mode/3`)
- Debug mode (see `debug_mode/3`)
- Exception mode (see `predicate_mode/3`)
- Modification mode (see `modify_mode/3`)

These attributes are retained even if the predicate is deleted (with `abolish/1` for example).

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

retract/1/2, dynamic/1, debug_mode/3, list_mode/3, modify_mode/3, predicate_mode/3

Return to the toplevel loop of IF/Prolog

abort

The predicate `abort/0` is a special predicate that neither succeeds nor fails. Its effect is purely procedural and causes normal execution to be interrupted.

Calling `abort/0` returns to the toplevel active input loop of the IF/Prolog system (`break/0`), regardless of how many such nested input loops are active. The *abort* message is output to the current error output stream.

If `abort/0` is called in the toplevel input loop, IF/Prolog remains in this loop. If there is no input loop active, IF/Prolog is exited with an error message.

Hints

The predicate `abort/0` is based on `throw/1`.

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Example

```
[user] ?- break.<ENTER>

[user][2] ?- break.<ENTER>

[user][3] ?- abort.<ENTER>
abort

[user] ?-
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

break/0, throw/1

Schedule alarm

alarm(+*Seconds* **)**

The predicate `alarm/1` causes the system to send the calling process an `alarm` signal after the number of real-time seconds specified by *Seconds* have elapsed. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If *Seconds* is negative or zero, a pending alarm request, if any, is cancelled.

Alarm requests are not stacked; only one `alarm` generation can be scheduled in this manner. If the `alarm` signal has not yet been generated, the call will result in rescheduling the time at which the `alarm` signal will be generated.

The current handling mode for the `alarm` can be queried with the predicate `current_signal/2` and changed with `signal_control/2`. It is recommendable to activate either the active (`on`) or the passive (`off`) signal handling mechanism.

Arguments

Seconds	Integer or integer expression

Exceptions

instantiation_error
The argument *Seconds* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Seconds* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

Furthermore, if an arithmetic expression is specified for the argument *Seconds*, all the exceptions for `is/2` can occur.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `alarm/1` is new.

See also

signal_handler/3, signal_control/2, context/2

Append or decompose lists

append(?*Head*, ?*Tail*, ?*List*)

The predicate `append/3` is used to append a term *Tail* (which is normally a list) to a list *Head* to form a complete list *List*.

There are four possible cases:

1. *Head* and *Tail* are instantiated;
   the predicate `append/3` appends *Tail* to *Head*. The result is unified with *List*.

2. *List* and one further argument are instantiated;
   the predicate `append/3` disengages *Head* or *Tail* from *List* and instantiates the result to the free argument.

3. Only *List* is instantiated;
   during backtracking, the predicate `append/3` generates all possible combinations of decomposing *List* into *Head* and *Tail*.

4. No argument or only *Head* or only *Tail* is instantiated;
   the predicate `append/3` generates variable masks (see last example).

Arguments

Head	List
Tail	List
List	List

Hints

Head must be a regular list (see section 9). If *Head* is a structure similar to a list (see page 812), the predicate fails.

If *Head* and *List* are uninstantiated variables, an infinite number of solutions may be generated by backtracking.

Under certain circumstances, *List* may be instantiated to any term. For example,

```
append([], term, term)
```

succeeds.

Example

This simply checks that appending `[a,b]` and `[c,d]` results in the list `[a,b,c,d]`.

```
[user] ?- append([a,b],[c,d],[a,b,c,d]).<ENTER>

yes
```

If the tail of the list `[a,b,c,d]` is instantiated to `[c,d]`, then `[a,b]` is the only possible head. There are no alternatives.

```
[user] ?- append(X,[c,d],[a,b,c,d]).<ENTER>

X       = [a,b] ;<ENTER>

no
```

One very important application of `append/3` is the generation of all possible ways of decomposing a list.

```
[user] ?- append(X,Y,[a,b]).<ENTER>

X       = []
Y       = [a,b] ;<ENTER>

X       = [a]
Y       = [b] ;<ENTER>

X       = [a,b]
Y       = [] ;<ENTER>

no
```

Although rarely used, it is permissible to append an atom or any other term to a linear list. The appended term appears as the tail in the result list.

```
[user] ?- append([a,b,c], f(a,b), X).<ENTER>

X       = [a,b,c|f(a,b)] <ENTER>

yes
```

This may mean that *List* is instantiated to any term.

```
[user] ?- append([],x + y, EL).<ENTER>

EL      = x + y <ENTER>

yes
```

The predicate `append/3` can also be used for generating variable masks.

```
[user] ?- append(X,Y,Z).<ENTER>

X       = []
Y       = _69
Z       = _69 ;<ENTER>

X       = [_125]
Y       = _69
Z       = [_125|_69] ;<ENTER>

X       = [_125,_129]
Y       = _69
Z       = [_125,_129|_69] ;<ENTER>

X       = [_125,_129,_133]
Y       = _69
Z       = [_125,_129,_133|_69] <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

connect/3, member/2

Access individual arguments of a structure

arg(+*Position*, +*Structure*, ?*Argument* **)**

The predicate `arg/3` unifies *Argument* with the term of *Structure* at the argument position specified by *Position*. If *Position* is 0 (zero) or greater than the arity of *Structure*, the predicate fails.

The predicate can be used to extract arguments from structures and to instantiate as yet uninstantiated arguments of structures (see examples).

Arguments

Position	Integer or integer expression
Structure	Structure
Argument	Term

Exceptions

instantiation_error
The argument *Position* or *Structure* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Position* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Position* must be a positive integer, but is negative.

type_error(compound)
The argument *Structure* must be a structure, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Position*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- arg(2,f(x,y,z),T).<ENTER>

T       = y <ENTER>

yes

[user] ?- S = struc(A,B),<ENTER>
          write(S),nl,<ENTER>
```

```
          arg(1,S,one),<ENTER>
          write(S),nl.<ENTER>
struc(_69,_70)
struc(one,_70)

S       = struc(one,_70)
A       = one
B       = _70 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *Position* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

functor/3, =../2

Test for array

array(@*TestTerm* **)**

The predicate `array/1` succeeds if *TestTerm* is an array term, otherwise it fails.

Arguments

TestTerm	Term

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `array/1` is new.

See also

array/2, create_array/2, get_array/3, set_array/3

Query dimension of an array

array(@*Array*, ?*Dimension* **)**

The predicate `array/2` queries the dimension of *Array* and unifies it with *Dimension*.

Arguments

Array	Array (system-defined term)
Dimension	Integer

Exceptions

instantiation_error
The argument *Array* must not be a variable, but a variable was specified.

type_error(array)
The argument *Array* must be an array, but is a term of another type.

type_error(integer)
The argument *Index* must be a variable or an integer, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `array/2` is new.

See also

array/1, create_array/2, get_array/3, set_array/3

Insert a clause in the database

asserta(@*Clause* **)** `[` @ +*Module* `]`

asserta(@*Head*, @*Body* **)** `[` @ +*Module* `]`

The predicate `asserta/1` inserts the specified *Clause* in the database in the calling module or in the specified *Module* as the new first clause of the associated predicate.

If the database does not contain the predicate in the module *Module* or if the Predicate is not visible in *Module*, the predicate will be created in *Module*.

The predicate `asserta/2` acts in the same way as `asserta/1`, but inserts the clause formed from *Head* and *Body*

Head `:-` *Body*

in the database.

Arguments

Clause	Structure, atom
Head	Structure, atom
Body	Term

Exceptions

instantiation_error
The argument *Head* or *Clause* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Head*, *Body* or *Clause* must have the syntactical structure of a Prolog goal.

permission_error(modify,static_procedure)
The argument *Head* or the head of *Clause* specifies an existing predicate which has not been declared as dynamic and therefore cannot be modified.

permission_error(access,calling_context)
The argument *Body* or the body of *Clause* contains a call involving `calling_context/1` in a predicate that is not a metapredicate.

Hints

The predicate `asserta/1/2` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

Standard

Only the predicate `asserta/1` complies with the definition of the ISO standard for Prolog.

See also

assertz/1/2, clause/2, retract/1/2, asserta_with_names/1/2, modify_mode/3

Insert a clause in the database

asserta_with_names(@*Clause*, @*VarList* **)** `[ @ +`*Module*` ]`

asserta_with_names(@*Head*, @*Body*, @*VarList* **)** `[ @ +`*Module*` ]`

The predicate `asserta_with_names/2` acts in the same way as `asserta/1`, but the information in the list *VarList* on variables in the clause is also stored in the database.

The predicate `asserta_with_names/3` acts in the same way as `asserta/2`, but the information in the list *VarList* on variable in the clause is also stored in the database.

The list *VarList* contains information on the variables occurring in the clause and their names in the following format:

`[` *Name* `=` *Variable*`, ... ]`

Only the information relating to the clause is transferred to the database. The information in *VarList* need not be complete.

Arguments

Clause	Structure, atom
Head	Structure, atom
Body	Term
VarList	List with format [*Name* = *Variable*,...]
Name	Atom
Variable	Variable

Exceptions

instantiation_error
: The argument *Head*, *Clause* or *VarList* must not be a variable, but a variable was specified.

type_error(callable)
: The argument *Head*, *Body* or *Clause* must have the syntactical structure of a Prolog goal.

permission_error(modify,static_procedure)
: The argument *Head* or the head of *Clause* specifies an existing predicate which has not been declared as dynamic and therefore cannot be modified.

permission_error(access,calling_context)
: The argument *Body* or the body of *Clause* contains a call involving `calling_context/1` in a predicate that is not a metapredicate.

type_error(list)
: The argument *VarList* must be a list, but is a term of another type.

type_error(assignment)
An element of *VarList* is not an assignment with the format *Name* = *Variable*.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

type_error(variable)
The argument *Variable* must be a variable, but is a term of another type.

Hints

The information about variables is used by `listing/1/2`, for example.

The predicate `asserta_with_names/2/3` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

assertz_with_names/2/3, clause_with_names/3, retract_with_names/2/3, asserta/1/2, modify_mode/3, listing/1/2

Insert a clause in the database

assertz(@*Clause* **)** `[ @ +`*Module*` ]`

assertz(@*Head*, @*Body* **)** `[ @ +`*Module*` ]`

The predicate `assertz/1` inserts the specified *Clause* in the database in the calling module or in the specified *Module* as the new last clause of the associated predicate.

If the database does not contain the predicate in the module *Module* or if the Predicate is not visible in *Module*, the predicate will be created in *Module*.

The predicate `assertz/2` acts in the same way as `assertz/1`, but inserts the clause formed from *Head* and *Body*

Head `:-` *Body*

in the database.

Arguments

Clause	Structure, atom
Head	Structure, atom
Body	Term

Exceptions

instantiation_error
The argument *Head* or *Clause* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Head*, *Body* or *Clause* must have the syntactical structure of a Prolog goal.

permission_error(modify,static_procedure)
The argument *Head* or the head of *Clause* specifies an existing predicate which has not been declared as dynamic and therefore cannot be modified.

permission_error(access,calling_context)
The argument *Body* or the body of *Clause* contains a call involving `calling_context/1` in a predicate that is not a metapredicate.

Hints

The predicates `consult/1` and `reconsult/1` use `assertz/2/3` to insert clauses in the database if the Prolog flag `debug` has the value `off`.

The predicate `assertz/1/2` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

Standard

Only the predicate `assertz/1` complies with the definition of the ISO standard for Prolog.

See also

asserta/1/2, clause/2, retract/1/2, assertz_with_names/1/2, modify_mode/3

Insert a clause in the database

assertz_with_names(@*Clause*, @*VarList* **)** `[` **@** `+`*Module* `]`

assertz_with_names(@*Head*, @*Body*, @*VarList* **)** `[` **@** `+`*Module* `]`

The predicate `assertz_with_names/2` acts in the same way as `assertz/1`, but the information in the list *VarList* on variables in the clause is also stored in the database.

The predicate `assertz_with_names/3` acts in the same way as `assertz/2`, but the information in the list *VarList* on variable in the clause is also stored in the database.

The list *VarList* contains information on the variables occurring in the clause and their names in the following format:

`[` *Name* `=` *Variable*`, ... ]`

Only the information relating to the clause is transferred to the database. The information in *VarList* need not be complete.

Arguments

Clause	Structure, atom
Head	Structure, atom
Body	Term
VarList	List with format [*Name* = *Variable*,...]
Name	Atom
Variable	Variable

Exceptions

instantiation_error
: The argument *Head*, *Clause* or *VarList* must not be a variable, but a variable was specified.

type_error(callable)
: The argument *Head*, *Body* or *Clause* must have the syntactical structure of a Prolog goal.

permission_error(modify,static_procedure)
: The argument *Head* or the head of *Clause* specifies an existing predicate which has not been declared as dynamic and therefore cannot be modified.

permission_error(access,calling_context)
: The argument *Body* or the body of *Clause* contains a call involving `calling_context/1` in a predicate that is not a metapredicate.

type_error(list)
: The argument *VarList* must be a list, but is a term of another type.

type_error(assignment)
An element of *VarList* is not an assignment with the format *Name* = *Variable*.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

type_error(variable)
The argument *Variable* must be a variable, but is a term of another type.

Hints

The predicates `consult/1` and `reconsult/1` use `assertz_with_names/2/3` to insert clauses in the database if the Prolog flag `debug` has the value `on`.

The information about variables is used by `listing/1/2`, for example.

The predicate `assertz_with_names/2/3` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

asserta_with_names/2/3, clause_with_names/3, retract_with_names/2/3, assertz/1/2, listing/1/2

Define an alias for an input/output stream

assign_alias(+*Alias*, @*Stream* **)**

The predicate `assign_alias/2` defines an alias *Alias* for the input/output stream *Stream*.

An alias is an alternative name that can be used as an argument for all input/output predicates instead of the system-defined term *Stream*. Any *Stream* may have more than one alias.

Arguments

Alias	Atom
Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Alias* or *Stream* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Alias* must be an atom, but is a term of another type.

permission_error(open,source_sink)
The argument *Alias* specifies an alias name that is already being used.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

Hints

Aliases may already be defined at `open/4`.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

cancel_alias/1, current_alias/2, open/3/4, set_input/1, set_output/1, set_error/1

Query end of line

at_end_of_line

at_end_of_line(@*Stream* **)**

The predicate `at_end_of_line/0` succeeds if the *current* input stream (`current_input`) is positioned at the end of a line.

The predicate `at_end_of_line/1` succeeds if the specified input stream *Stream* is positioned at the end of a line.

An input stream is positioned at the end of an input line if a subsequent character read operation would yield the new line character '`\n`' or if the end of the input stream would be reached.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

Hints

The predicates `at_end_of_line/0/1` must inspect the next character in the input stream. It is therefore possible that the system will have to wait for an input.

Example

The following predicate skips the rest of an input line, including the terminating new line character:

```
[user] ?- listing.<ENTER>

% *** user: skip_comment / 0 ***
skip_comment :-
    repeat,
    get_char(_),
    at_end_of_line,
    get_char(_),
    ! .

yes

[user] ?- read(T), skip_comment, read(N).<ENTER>
word. Comment<ENTER>
new_line.<ENTER>

T       = word
N       = new_line <ENTER>

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

skip_line/0/1, at_end_of_stream/0/1

Query end of stream

at_end_of_stream

at_end_of_stream(@*Stream*)

The predicate `at_end_of_stream/0` succeeds if the *current* input stream (`current_input`) is positioned at the end.

The predicate `at_end_of_stream/1` succeeds if the specified input stream *Stream* is positioned at the end.

An input stream is positioned at the end if a subsequent character read operation would report the end of the input stream.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

Hints

The predicates `at_end_of_stream/0/1` must perform a read operation in the input stream to determine whether the end of the stream has been reached. It is therefore possible that the system will have to wait for an input.

Example

The following predicate skips the rest of an input stream:

```
[user] ?- listing.<ENTER>

% *** user: skip_rest / 1 ***
skip_rest(Stream) :-
    repeat,
    get_char(Stream,_),
    at_end_of_stream(Stream),
    ! .

yes

[user] ?- read(user_input,T),<ENTER>
          skip_rest(user_input),<ENTER>
          read(user_input,N).<ENTER>
one_before_last_line. Comment<ENTER>
last_line. Comment<ENTER>
<EOF>

T       = one_before_last_line
N       = end_of_file <ENTER>

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

See also

at_end_of_line/0/1

Test for atom

atom(@*TestTerm* **)**

The predicate `atom/1` succeeds if *TestTerm* is an atom, otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- atom(atom).<ENTER>

yes

[user] ?- atom(X).<ENTER>

no

[user] ?- X = atom, atom(X).<ENTER>

X       = atom <ENTER>

yes

[user] ?- atom([]).<ENTER>

yes

[user] ?- atom( f(a,B)).<ENTER>

no

[user] ?- atom( 'f(a,B)').<ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atomic/1, compound/1, cyclic/1, float/1, ground/1, integer/1, nonvar/1, number/1, rational/1, var/1

Convert between an atom and individual characters

atom_chars(+*Atom*, ?*List* **)**
atom_chars(-*Atom*, @*List* **)**

The predicate `atom_chars/2` establishes the relationship between an atom and the list comprising of the individual characters of the atom. A distinction is made between two cases:

1. *Atom* is instantiated;
   List must be either a variable or a list of variables or characters. The predicate `atom_chars/2` creates the list of characters of *Atom* and unifies the result with *List*.

2. *Atom* is uninstantiated;
   List must be a completely instantiated list of characters. The predicate `atom_chars/2` uses *List* to create an atom, the characters of which correspond to the characters in *List* and unifies the result with *Atom*.

Arguments

Atom	Atom
List	List of characters

Exceptions

instantiation_error
The argument *List* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(character)
An element of *List* must be a character, but is a term of a another type.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

Example

```
[user] ?- atom_chars(atom,X).<ENTER>

X       = [a,t,o,m] <ENTER>

yes

[user] ?- atom_chars(X,[a,t,o,m]).<ENTER>
```

```
X       = atom <ENTER>

yes
```

List may be partially uninstantiated if *Atom* is instantiated.

```
[user] ?- atom_chars(abcd,[_,X,c,Y]).<ENTER>

X       = b
Y       = d <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

number_chars/2, atom_codes/2

Convert between an atom and individual character codes

atom_codes(+*Atom*, **?***List*)
atom_codes(-*Atom*, **@***List*)

The predicate `atom_codes/2` establishes the relationship between an atom and the list comprising of the individual character codes of the atom. A distinction is made between two cases:

1. *Atom* is instantiated;
   List must be either a variable or a list of variables or character codes. The predicate `atom_codes/2` creates the list of character codes of *Atom* and unifies the result with *List*.

2. *Atom* is uninstantiated;
   List must be a completely instantiated list of character codes. The predicate `atom_codes/2` uses *List* to create an atom, the characters of which correspond to the character codes in *List* and unifies the result with *Atom*.

Arguments

Atom	Atom
List	List of integers or integer expressions

Exceptions

instantiation_error
The argument *List* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(integer)
An element of *List* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

representation_error(character_code)
An element of *List* must be an integer representing a character code.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

Furthermore, if an arithmetic expression is specified for an element in *List*, all the exceptions for `is/2` can occur.

Hints

Integer expressions will only be evaluated if *Atom* is not instantiated.

Example

```
[user] ?- atom_codes(hello,X).<ENTER>

X       = [104,101,108,108,111] <ENTER>

yes

[user] ?- atom_codes(X,[97,116,111,109]).<ENTER>

X       = atom <ENTER>

yes
```

List may be partially uninstantiated if *Atom* is instantiated.

```
[user] ?- atom_codes(abcd,[_,X,99,Y]).<ENTER>

X       = 98
Y       = 100 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *List* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

number_codes/2, atom_chars/2, char_code/2

Compose or decompose an atom

atom_concat(?*Atom1*, ?*Atom2*, +*Atom3* **)**
atom_concat(+*Atom1*, +*Atom2*, -*Atom3* **)**

The predicate `atom_concat/3` succeeds, if *Atom3* is a concatenation of *Atom1* and *Atom2*. A distinction is made between the following two cases:

1. *Atom3* is instantiated;
   Atom1 and *Atom2* are unified by backtracking with all the possible decompositions of *Atom3*.

2. *Atom3* is uninstantiated;
   Atom3 is unified with the atom arising from the concatenation of *Atom1* and *Atom2*.

Arguments

Atom1	Atom
Atom2	Atom
Atom3	Atom

Exceptions

instantiation_error
The argument *Atom1*, *Atom2* and *Atom3* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom1*, *Atom2* or *Atom3* must be an atom, but is a term of another type.

Example

```
[user] ?- atom_concat(abc,def,A).<ENTER>

A       = abcdef <ENTER>

yes

[user] ?- atom_concat(A,B,abc).<ENTER>

A       = ''
B       = abc ;<ENTER>
```

```
A       = a
B       = bc ;<ENTER>

A       = ab
B       = c ;<ENTER>

A       = abc
B       = '' ;<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

concat_atom/2/3, atom_part/4, atom_prefix/3, atom_suffix/3, getchar/3, index/3, sub_atom/5, atom_split/3

Count the characters in an atom

atom_length(+*Atom*, ?*Length* **)**

The predicate `atom_length/2` unifies *Length* with the number of characters in the atom *Atom*.

Arguments

Atom	Atom
Length	Integer

Exceptions

instantiation_error
The argument *Atom* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(integer)
The argument *Length* must be a variable or an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Length* must be a positive integer, but is negative.

Example

The atom *short* is 5 characters long.

```
[user] ?- atom_length(short,5).<ENTER>

yes
```

Single quotes used in the external representation are not part of an atom.

```
[user] ?- atom_length('ATOM',X).<ENTER>

X       = 4 <ENTER>

yes

[user] ?- atom_length('',X).<ENTER>

X       = 0 <ENTER>
```

```
yes

[user] ?- atom_length([],X).<ENTER>

X       = 2 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom_concat/3, atom_part/4, atom_suffix/3, atom_prefix/3, getchar/3, index/3, sub_atom/5, byte_length/2

Convert between an atom and a number

atom_number(+*Atom*, **?***Number* **)**
atom_number(-*Atom*, **@***Number* **)**

The predicate `atom_number/2` establishes the relationship between an atom and a number. A distinction is made between two cases:

1. *Atom* is instantiated;
   Number must be either a variable or a number. The predicate `atom_number/2` converts *Atom* to a number and unifies the result with *Number*.

2. *Atom* is uninstantiated;
   Number must be a number or an arithmetic expression. The predicate `atom_number/2` converts *Number* to an atom, and unifies the result with *Atom*.

Arguments

Atom	Atom
Number	Number

Exceptions

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

representation_error(number)
The argument *Atom* must be an atom that is representable as a number.

instantiation_error
The argument *Number* or a subterm must not be a variable, but a variable was specified.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

Hints

Arithmetic expressions will only be evaluated if *Atom* is not instantiated.

Example

```
[user] ?- atom_number('140218',X).<ENTER>
```

```
X       = 140218 <ENTER>

yes

[user] ?- atom_number(X,10/7).<ENTER>

X       = '1.4285714285714286' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate generates an exception if the atom is not representable as a number.

See also

atom_codes/2, atom_chars/2, number_codes/2, number_chars/2

Determine the subatom of an atom

atom_part(+*Atom*, +*Position*, +*Length*, ?*SubAtom* **)**

The predicate `atom_part/4` unifies *SubAtom* with the part of *Atom* defined by *Position* and *Length*.

If *Position* is less than 1 or greater than the length of *Atom*, then *SubAtom* is unified with the null atom `''`. The same applies if *Length* is less than 1.

If *Length* is greater than the remaining length of *Atom* after *Position*, then *SubAtom* is unified with the rest of *Atom* starting from *Position*.

Arguments

Atom	Atom
Position	Integer or integer expression
Length	Integer or integer expression
SubAtom	Atom

Exceptions

instantiation_error
The argument *Atom*, *Position* or *Length* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(integer)
The argument *Position* or *Length* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(atom)
The argument *SubAtom* must be a variable or an atom, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Position* or *Length*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- atom_part(atom_part,6,4,P).<ENTER>

P       = part <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom_concat/3, atom_prefix/3, atom_suffix/3, getchar/3, index/3, sub_atom/5, atom_split/3

Determine the prefix of an atom

atom_prefix(+*Atom*, +*Length*, ?*Prefix* **)**

The predicate `atom_prefix/3` unifies *Prefix* with the atom formed by the first *Length* characters in *Atom*.

If *Length* is less than 1, *Prefix* is unified with the null atom `''`.

If *Length* is greater than the length of *Atom*, *Prefix* is unified with *Atom*.

Arguments

Atom	Atom
Length	Integer or integer expression
Prefix	Atom

Exceptions

instantiation_error
The argument *Atom* or *Length* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(integer)
The argument *Length* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(atom)
The argument *Prefix* must be a variable or an atom, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Length*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- atom_prefix('file.pro',4,B).<ENTER>

B       = file <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom_concat/3, atom_suffix/3, atom_part/4, getchar/3, index/3, sub_atom/5, atom_split/3

Decompose an atom

atom_split(+*Atom*, +*Delimiter*, ?*Subatoms* **)**

The predicate `atom_split/3` splits *Atom* into a list of subatoms *Subatoms*. *Delimiter* is the delimiter of the subatoms and will not appear in *Subatoms*.

Arguments

Atom	Atom
Delimiter	Atom
Subatoms	List of atoms

Exceptions

instantiation_error
: The argument *Atom* or *Delimiter* must not be a variable, but a variable was specified.

type_error(atom)
: The argument *Atom* or *Delimiter* must be an atom, but is a term of another type.

type_error(list)
: The argument *Subatoms* must be a list, but is a term of another type.

type_error(atom)
: An element of *Subatoms* must be a variable or an atom, but is a term of another type.

Hints

The predicate is the opposite of the predicate `concat_atom/3`.

Example

```
[user] ?- atom_split('/var/adm/log/FSCK', '/', L).<ENTER>

L       = ['',var,adm,log,'FSCK']

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `atom_split/3` is new.

See also

concat_atom/2/3, atom_concat/3, atom_part/4, atom_prefix/3, atom_suffix/3, getchar/3, index/3, sub_atom/5

Determine the suffix of an atom

atom_suffix(+*Atom*, +*Length*, ?*Suffix* **)**

The predicate `atom_suffix/3` unifies *Suffix* with the atom formed by the last *Length* characters in *Atom*.

If *Length* is less than 1, *Suffix* is unified with the null atom `''`.

If *Length* is greater than the length of *Atom*, *Suffix* is unified with *Atom*.

Arguments

Atom	Atom
Length	Integer or integer expression
Suffix	Atom

Exceptions

instantiation_error
The argument *Atom* or *Length* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(integer)
The argument *Length* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(atom)
The argument *Suffix* must be a variable or an atom, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Length*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- atom_suffix('file.pro',3,S).<ENTER>

S       = pro <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom_concat/3, atom_prefix/3, atom_part/4, getchar/3, index/3, sub_atom/5, atom_split/3

Test for constant

atomic(@*TestTerm* **)**

The predicate `atomic/1` succeeds if *TestTerm* is a constant (atom or number), otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- atomic( 5.240551 ).<ENTER>

yes

[user] ?- X is 5 - 2, atomic( X ).<ENTER>

X       = 3 <ENTER>

yes

[user] ?- atomic( f(a,b) ).<ENTER>

no

[user] ?- atomic( X ).<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom/1, compound/1, cyclic/1, float/1, ground/1, integer/1, nonvar/1, number/1, rational/1, var/1

Find selected solutions of a goal

bagof(?*Term*, +*Goal*, ?*TermList* **)** `[` **@** +*Module* `]`

The predicate `bagof/3` unifies *TermList* with a list of all instantiations of *Term*, with which *Goal* succeeds.

Term and *Goal* should share common variables.

Backtracking can take place by the variables which occur in *Goal* and *not* in *Term* (known as *free* variables). If `bagof/3` succeeds, the variables occurring both in *Term* and in *Goal* remain uninstantiated.

If *Goal* is specified as

Term1 ^ *Term2* ^ ... ^ *SubGoal*

SubGoal is executed and the variables in *Term1*, *Term2* are no longer considered free, even if they occur only in *Goal* and not in *Term*. Backtracking, as described above, does not therefore take place by these variables.

Arguments

Term	Term
Goal	Goal
TermList	List
*Term*i	Term
SubGoal	Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

type_error(list)
The argument *TermList* must be a variable or a list, but is a term of another type.

Hints

The atom `'^'` is defined as an infix operator.

The predicate `bagof/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- bagof(X,current_op(V,fy,X),L).<ENTER>

X       = _68
V       = 200
L       = [\,-,+] ;<ENTER>

X       = _68
V       = 900
L       = [\+,not] ;<ENTER>

no

[user] ?- bagof(X,(X=1;X=2;X=1),S).<ENTER>

X       = _152
S       = [1,2,1] <ENTER>

yes

[user] ?- bagof(X,Y^((X=1,Y=1);(X=2,Y=2);(X=1,Y=3)),S).<ENTER>

X       = _152
S       = [1,2,1] <ENTER>
```

```
yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

call/1, findall/3, setof/3, @/2, :/2, set_prolog_flag/2

Define the body of a module

:- begin_module +*Module*
:- begin_module(+*Module* **)**

The directive `begin_module/1` marks the start of the body of the module *Module*. The body contains a declaration part containing directives (e.g. `initialization/1` or `import/1/2`), and the implementation part of the module, i.e. clauses of the predicates.

At the time the directive is processed, the interface of the module must be known to IF/Prolog. This applies, if the interface of *Module* was defined in the Prolog text *before* the directive `begin_module/1` or the interface was loaded into IF/Prolog before the directive was processed.

The body of one module ends with the start of a new module (`module/1`), at the start of another body (`begin_module/1`), at the start of the `end_module/0/1` directive or at the end of a file.

If a body has already been loaded for *Module*, it is completely deleted as soon the `begin_module/1` directive appears.

Arguments

Module	Atom, name of a module

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *Module* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

existence_error(module)
The argument *Module* is not the name of an existing module.

permission_error(modify,system_module)
The argument *Module* is the name of a system module. System modules may not be modified.

Hints

The atom `begin_module` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive *Module* is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, end_module/0/1, import/1/2, export/1, reexport/1/2, meta/1

Generate a new input loop

break

The predicate `break/0` generates a new input loop of IF/Prolog.

Thus input loops can be nested. The input prompt shows the current nesting depth for input loops from level 2 onward, e.g.
`[user][2] ?-`
This predicate enables the user to suspend the current goal and enter a dialog with IF/Prolog. When the new input loop is terminated, the suspended goal is resumed.

An input loop is terminated when the atom `end_of_file` is entered or when an equivalent key (e.g. CTRL D) is pressed. `abort/0` returns to the toplevel input loop of IF/Prolog.

Hints

The input loop of IF/Prolog defines a context (see `context/2`) which intercepts **all** the exceptions that are not handled directly. The occurrence of errors or exceptions does not therefore lead to the abandonment of the IF/Prolog input loop.

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Example

```
[user] ?- break.<ENTER>

[user][2] ?- <EOF>

[user] ?-
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

abort/0, set_default_module/1, current_default_module/1

Count the bytes in an atom

byte_length(+*Atom*, ?*Length* **)**

The predicate `byte_length/2` unifies *Length* with the number of bytes in the atom *Atom*.

The predicate is equivalent to `atom_length/2`, if IF/Prolog has not been configured with multiple-octet characters or the current language is `ascii`.

Arguments

Atom	Atom
Length	Integer

Exceptions

instantiation_error
The argument *Atom* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(integer)
The argument *Length* must be a variable or an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Length* must be a positive integer, but is negative.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom_length/2

Call a goal

call(+*Goal* **)** `[` **@** +*Module* `]`

The language construct `call/1` calls the goal *Goal* in the calling module or in the specified *Module*, and succeeds if *Goal* succeeds.

By backtracking, *Goal* can return several results.

If *Goal* is a composite goal and contains a **cut** (`!/0`), then this cut will have no effect beyond `call/1`.

Arguments

Goal	Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The language construct `call/1` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `:/2` is not permitted for this language construct.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of the calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- !, fail; true.<ENTER>

no

[user] ?- call(!), fail; true.<ENTER>

yes
```

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

call/2, once/1, @/2, :/2, set_prolog_flag/2

Call a goal

call(+*Name*, +*ArgList* **)** `[` **@** +*Module* `]`

The language construct `call/2` constructs a goal from *Name* and *ArgList*:

Goal `=..` `[` *Name* `|` *ArgList* `]`

The constructed goal *Goal* is executed in the calling module or in the specified *Module*. The predicate `call/2` succeeds if *Goal* succeeds.

By backtracking, the goal can return several results.

Arguments

Name	Atom
ArgList	List

Exceptions

instantiation_error
The argument *Name* or *List* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

type_error(list)
The argument *List* must be a list, but is a term of another type.

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The language construct `call/2` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `:/2` is not permitted for this language construct.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of the calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- call(member, [A, [a,b]]).<ENTER>

A       = a ;<ENTER>

A       = b ;<ENTER>

no
```

Standard

This language construct is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

call/1, once/1, @/2, :/2, set_prolog_flag/2

Query the module argument of metapredicates

calling_context(?*Module* **)**

The control construct `calling_context/1` queries the module argument of metapredicates. The module argument is the argument that is made available explicitly or implicitly to metapredicates as an extra argument. As a rule, this argument is normally given the name of the module from which the metapredicate was called.

Arguments

Module	Atom, name of a module

Exceptions

type_error(atom)
: The argument *Module* must be a variable or an atom, but is a term of another type.

Standard

This language construct is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

@/2, meta/1

Cancel an alias for an input/output stream

cancel_alias(+*Alias* **)**

The predicate `cancel_alias/1` cancels the alias *Alias* of an input/output stream.

This is done automatically when an input/output stream is closed (see `close/1`).

The standard aliases (see `current_alias/2`) cannot be canceled, but `cancel_alias/1` succeeds nevertheless. Likewise, `cancel_alias/1` succeeds, even if *Alias* is not a defined alias.

Arguments

Alias	Atom

Exceptions

instantiation_error
The argument *Alias* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Alias* must be an atom, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

assign_alias/2, current_alias/2

Intercept a context jump

catch(+*Goal*, +*CatchMask*, +*SubstituteGoal* **)** `[ @` +*Module* `]`

The predicate `catch/3` opens a context for intercepting context jumps and calls the goal *Goal*.

If *Goal* succeeds, `catch/3` is successfully exited. By backtracking, *Goal* can return several results.
If *Goal* does not (or does no longer) succeed, `catch/3` also fails (i.e. backtracking is initiated).

If `throw(`*Term*`)` is called while *Goal* is being executed, execution is terminated and *CatchMask* is unified with a copy of *Term*. If the unification is successful, *SubstituteGoal* is called. If the unification fails, the next outer context is used (see `throw/1`).

Exceptions which are detected by IF/Prolog and which are not processed using a handler (see `context/2`),

`throw(error(`*Error*`,`*Info*`))`

is implicitly called, where *Error* indicates the error type and *Info* is a list of terms of the type *InfoName* = *Value*, which provide additional information about the error. *InfoName* is an atom indicating the type of additional information and *Value* is any term (see also `exception/2`).

Arguments

Goal	Goal
CatchMask	Term
SubstituteGoal	Goal

Exceptions

instantiation_error
: The argument *Goal* or *SubstituteGoal* or a subterm must not be a variable, but a variable was specified.

instantiation_error
: The argument *Goal* or *SubstituteGoal* must not be a variable, but a variable was specified.

type_error(callable)
: The argument *Goal* or *SubstituteGoal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
: In executing *Goal* or *SubstituteGoal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or *SubstituteGoal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or *SubstituteGoal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate `catch/3` is based on `context/2`.

The predicate `catch/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

call/1, throw/1, context/2, exception/2, set_prolog_flag/2

Convert between a character and a character code

char_code(+*Character*, ?*CharCode* **)**
char_code(-*Character*, +*CharCode* **)**

The predicate `char_code/2` converts a *Character* into a *CharCode* or vice versa. The following cases are possible:

1. *Character* is instantiated;
   the character code of *Character* is determined and unified with *CharCode*.

2. *Character* is uninstantiated;
   the character corresponding to *CharCode* is formed and unified with *Character*.

Arguments

Character	Character
CharCode	Integer or integer expression

Exceptions

type_error(character)
The argument *Character* must be a character, but is a term of another type.

representation_error(character)
The argument *Character* is not representable as a single character.

instantiation_error
The argument *CharCode* must not be a variable, but a variable was specified.

type_error(integer)
The argument *CharCode* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

representation_error(character_code)
The argument *CharCode* must be an integer representing a character code.

Furthermore, if an arithmetic expression is specified for the argument *CharCode*, all the exceptions for `is/2` can occur.

Hints

Integer expressions for *CharCode* will only be evaluated if *Character* is not instantiated.

Example

```
[user] ?- char_code( X, 98).<ENTER>

X       = b <ENTER>

yes

[user] ?- char_code('A',Y).<ENTER>

Y       = 65 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *CharCode* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

atom_codes/2

Define a character conversion

char_conversion(+*CharacterIn*, +*CharacterOut* **)**

The predicate `char_conversion/2` defines a character conversion.

A **character conversion** specifies that, when reading terms, each character *CharacterIn* which is not enclosed in single or double quotes or which is used in a character constant is to be replaced by the character *CharacterOut*.

Character conversion only takes place if the Prolog flag `char_conversion` is set to `on`. Otherwise, character conversions are stored but do not affect the reading of terms.

The definition of a character conversion is global, i.e. it is not limited to the reading of *one* Prolog text.

Arguments

CharacterIn	Character
CharacterOut	Character

Exceptions

instantiation_error
: The argument *CharacterIn* or *CharacterOut* must not be a variable, but a variable was specified.

type_error(character)
: The argument *CharacterIn* or *CharacterOut* must be a character, but is a term of another type.

representation_error(character)
: The argument *CharacterIn* or *CharacterOut* is not representable as a single character.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

current_char_conversion/2

Define a character conversion (directive)

:- char_conversion(+*CharacterIn*, +*CharacterOut* **)**

The directive `char_conversion/2` defines a character conversion. It functions in exactly the same way as the predicate `char_conversion/2`. As as directive, it is executed by `consult/1`, `reconsult/1` or `compile/1` when a Prolog source text is read and takes effect immediately.

Arguments

CharacterIn	Character
CharacterOut	Character

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
: The argument *CharacterIn* or *CharacterOut* must not be a variable, but a variable was specified.

type_error(character)
: The argument *CharacterIn* or *CharacterOut* must be a character, but is a term of another type.

representation_error(character)
: The argument *CharacterIn* or *CharacterOut* is not representable as a single character.

Hints

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Character conversions take effect immediately as soon as the directive is evaluated, provided that the Prolog flag `char_conversion` is set to `on`.

Standard

The directive `char_conversion/2` complies with the definition of the ISO standard for Prolog.

See also

current_char_conversion/2

Change current directory

chdir(*+Dir* **)**

The predicate `chdir/1` changes the current directory for the IF/Prolog process. The predicate succeeds if *Dir* can be made the current directory, otherwise it fails.

Arguments

Dir	Atom

Exceptions

instantiation_error
The argument *Dir* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Dir* must be an atom, but is a term of another type.

Example

```
[user] ?- chdir('/usr/bin'),<ENTER>
          system(pwd).<ENTER>
/usr/bin

yes

[user] ?- chdir('..'),<ENTER>
          getcwd(Pwd).<ENTER>

X = '/usr'<ENTER>

yes

[user] ?- chdir('hello').<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

getcwd/1, windows_chdrive/1, windows_getdrive/1

Search the database for specific clauses

clause(+*Head*, ?*Body* **)** `[` **@** +*Module* `]`

The predicate `clause/2` searches the database in the calling module or in the specified *Module* for clauses with a head which can be unified with *Head*. If such a clause is found, unification of *Head* is performed, i.e. any variables in *Head* are unified with the corresponding arguments from the head of the clause found and *Body* is unified with the body of the clause found.

The predicate `clause/2` succeeds by backtracking for all matching clauses which were found in the database at the time of the first call, even if they have since been modified (**logic update view**).

Arguments

Head	Structure, atom
Body	Term

Exceptions

instantiation_error
The argument *Head* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Head* must have the syntactical structure of a clause head.

permission_error(access,static_procedure)
The argument *Head* specifies a predicate that was not declared as dynamic.

type_error(callable)
The argument *Body* must have the syntactical structure of a Prolog goal.

Hints

Facts in the database have the clause body `true`.

The predicate `clause/2` is a metapredicate and considers only predicate definitions which are visible in the calling module or in the specified *Module*.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

asserta/1/2, assertz/1/2, clause_with_names/3

Search the database for specific clauses

clause_with_names(+*Head*, ?*Body*, ?*VarList* **)** `[` **@** +*Module* `]`

The predicate `clause_with_names/3` acts in the same way as `clause/2`, except that any name information on variables is also unified with *VarList* in the form:

`[` *Name* `=` *Variable*`, ... ]`

If there is no such information, *VarList* is unified with the empty list.

The predicate `clause_with_names/3` succeeds by backtracking for all matching clauses which were found in the database at the time of the first call, even if they have since been modified (**logic update view**).

Arguments

Head	Structure, atom
Body	Term
VarList	List in the form [*Name* = *Variable*,...]
Name	Atom
Variable	Variable

Exceptions

instantiation_error
The argument *Head* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Head* must have the syntactical structure of a clause head.

permission_error(access,static_procedure)
The argument *Head* specifies a predicate that was not declared as dynamic.

type_error(callable)
The argument *Body* must have the syntactical structure of a Prolog goal.

type_error(list)
The argument *VarList* must be a variable or a list, but is a term of another type.

type_error(assignment)
An element of *VarList* is not an assignment with the format *Name* `=` *Variable*.

type_error(atom)
The argument *Name* must be a variable or an atom, but is a term of another type.

type_error(variable)
The argument *Variable* must be a variable, but is a term of another type.

Hints

The predicate `clause_with_names/3` is a metapredicate and considers only predicate definitions which are visible in the calling module or in the specified *Module*.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

clause/2, asserta_with_names/2/3, assertz_with_names/2/3, listing/0/1

Clear to end of line

clear_eol

clear_eol(@*Stream* **)**

The predicate `clear_eol/0` clears the remainder of the line on the screen connected to the current output stream, starting at the cursor position. The position where the cursor is currently located is also cleared.

If there is no screen connected to the current output stream or if the device driver for this output stream is not capable of executing this operation, the predicate has no effect and the call succeeds nevertheless.

The predicate `clear_eol/1` acts in the same way as `clear_eol/0`, but uses the screen connected to the specified output stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

Hints

The terminal control predicates are implemented via the appropriate stream control functions of the device drivers (see `stream_control/2`).

The standard device drivers implement screen control via the **terminfo/termcap** service of the operating system. They support only one terminal type for all output streams.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

clear_eos/0/1, clear_screen/0/1

Clear to end of screen

clear_eos

clear_eos(@*Stream*)

The predicate `clear_eos/0` clears the remainder of the screen connected to the current output stream, starting at the cursor position. The position where the cursor is currently located is also cleared.

If there is no screen connected to the current output stream or if the device driver for this output stream is not capable of executing this operation, the predicate has no effect and the call succeeds nevertheless.

The predicate `clear_eos/1` acts in the same way as `clear_eos/0`, but uses the screen connected to the specified output stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

Hints

The terminal control predicates are implemented via the appropriate stream control functions of the device drivers (see `stream_control/2`).

The standard device drivers implement screen control via the **terminfo/termcap** service of the operating system. They support only one terminal type for all output streams.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

clear_eol/0/1, clear_screen/0/1

Clear screen

clear_screen

clear_screen(@*Stream*)

The predicate `clear_screen/0` clears the entire screen connected to the current output stream. The cursor is placed in the top left-hand corner of the screen.

If there is no screen connected to the current output stream or if the device driver for this output stream is not capable of executing this operation, the predicate has no effect and the call succeeds nevertheless.

The predicate `clear_screen/1` acts in the same way as `clear_screen/0`, but uses the screen connected to the specified output stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

Hints

The terminal control predicates are implemented via the appropriate stream control functions of the device drivers (see `stream_control/2`).

The standard device drivers implement screen control via the **terminfo/termcap** service of the operating system. They support only one terminal type for all output streams.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

clear_eol/0/1, clear_eos/0/1

Close an input/output stream

close(@*Stream* **)**

close(@*Stream*, @*Actions* **)**

The predicate `close/1` closes the input/output stream *Stream*. Beforehand, any buffer contents for an output stream are flushed.

All aliases associated with this *Stream* are deleted.

If *Stream* is the current input, output or error output stream, it is reset to `user_input`, `user_output` or `user_error`.

The default streams opened by IF/Prolog cannot be closed, but `close/1` succeeds nevertheless.

The predicate `close/2` acts in the same way as `close/1`, except that the specified *Actions* are considered.

The list *Actions* may contain the following elements:

`force(false)` If it is not possible to close the *Stream* properly, it should remain open.

`force(true)` If it is not possible to close the *Stream* properly, this should be forced, even if it means a loss of data (default).

Arguments

Stream	Stream (system-defined term) or alias (atom)
Actions	List of structures

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

instantiation_error
The argument *Actions* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *Actions* must be a list, but is a term of another type.

domain_error(close_option)
The argument *Actions* specifies an option that is not permitted for `close/2`.

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Hints

It depends on the device as to whether specifying *Actions* has an effect on the behavior of the device driver. Often it will be impossible that the stream **cannot** be closed successfully.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

open/3/4

Compare terms

compare(?*Op*, @*Term1*, @*Term2*)

The predicate `compare/3` checks the relationship between *Term1* and *Term2*.

The argument *Op* is unified with one of the following atoms:

= The terms are identical

< *Term1* is smaller than *Term2*

> *Term1* is greater than *Term2*

The following relationships apply to terms of different types:

Variable	<	Floating point
Floating point	<	Rational number
Rational number	<	Integer
Integer	<	Atom
Atom	<	Structure

The following applies to terms of the same type:

Variable The order is undefined and the result is therefore random.

Floating point Arithmetic order.

Rational number Arithmetic order.

Integer Arithmetic order.

Atom Character code order.

Structure Ordered primarily according to their arity, then by the name of the functor, and finally by the arguments from left to right.

Arguments

Op	Atom: = | < | >
Term1	Term
Term2	Term

Exceptions

type_error(atom)
The argument *Op* must be a variable or an atom, but is a term of another type.

Example

```
[user] ?- compare(Op, adam, zeppelin).<ENTER>

Op      = < <ENTER>

yes

[user] ?- compare(<, 3.77, 1).<ENTER>

yes

[user] ?- compare(>, 13, hello).<ENTER>

no

[user] ?- compare(Op, g(z), g(x,y)).<ENTER>

Op      = < <ENTER>

yes

[user] ?- X is 4*5, Y is 2*10, compare(Op, X, Y).<ENTER>

X       = 20
Y       = 20
Op      = = <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.0B The predicate `compare/3` is new.

See also

@</2, @=</2, @>/2, @>=/2, @=/2, @\=/2

Compile a Prolog text

compile(+*Filename* **)**

The predicate `compile/1` compiles the Prolog text in the file with the name *Filename*.

If *Filename* does not contain the extension *.pro* and if a file of this name cannot be opened for reading, the extension is appended and the resultant filename is used.

The compiled Prolog text is stored in internal format in a file with the extension *.cmp*. This file is then implicitly loaded with the predicate `load/1`.

If *Filename* contains only module bodies, the interface of this module must be known to IF/Prolog for compilation, and the complete module is stored in the file created.

In contrast to incremental compilation used with `asserta/1/2`, etc., additional optimizations can be made with `compile/1`. All the predicates defined in *Filename* which have not been declared with the `dynamic/1` directive can then no longer be modified, displayed or tested. This applies irrespective of the value of the Prolog flag `debug`.

Arguments

Filename	Atom, name of a file

Exceptions

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The argument *Filename* specifies a file for which the indicated error was reported by the operating system during file processing.

existence_error(past_end_of_stream)
The argument *Filename* specifies a file for which an unexpected end-of-file was detected during file processing.

In addition, all the error messages and warnings from `read_term/2`, `assertz/1` and `assertz_with_names/2` and may occur. These do not cause the load process to be aborted. You should be aware that the database may not necessarily contain what you expect it to contain.

Errors in directives are also output as error messages on the error output stream. Unknown directives are reported by the error message `domain_error(directive)`.

In the event of errors in a module interface, the affected module is deleted after *Filename* has been processed.

If there are errors in the declaration part of a module body, the affected module body is deleted after *Filename* has been processed.

When initialization directives are executed (see `init/1`), any exception may arise. These aborts the execution of the initialization goal, but does not abort the load process.

In addition, warnings are output as a function of the Prolog flag `warnings` in certain situations to alert you to possible sources of errors.

Hints

The name of the file which contains the compiled Prolog text is *Filename*, with the extension *.pro* being replaced by the extension *.cmp*; if there is no extension, *.cmp* is simply added to the filename.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

load/1, unload/1, save_module/1

Test for structure

compound(@TestTerm)

The predicate `compound/1` succeeds if *TestTerm* is a structure, otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- compound(a(_,_)).<ENTER>

yes

[user] ?- compound(a+b).<ENTER>

yes

[user] ?- compound('.'(a,b)).<ENTER>

yes

[user] ?- compound(a).<ENTER>

no

[user] ?- X = *(a,+(b,/(c,**(d,-(e,f))))),<ENTER>
          compound(X).<ENTER>

X       = a * (b + c / d ** (e - f)) <ENTER>

yes

[user] ?- compound( not true ).<ENTER>

yes

[user] ?- compound([a,b]).<ENTER>

yes
```

The empty list is not a structure (it is an atom).

```
[user] ?- compound([]).<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom/1, atomic/1, cyclic/1, float/1, ground/1, integer/1, nonvar/1, number/1, rational/1, var/1

Concatenate individual atoms to form an atom

concat_atom(@*List*, ?*Atom* **)**

concat_atom(@*List*, +*Delimiter*, ?*Atom* **)**

The predicates `concat_atom/2/3` concatenate the elements specified in *List* to form an atom and unify this with *Atom*. If integers occur in *List*, they are converted beforehand into atoms (see `write_atom/2`).

The predicate `concat_atom/3` acts in the same way as `concat_atom/2`, but inserts the atom *Delimiter* between every pair of elements in *List*.

Arguments

List	List of atoms and/or integers
Delimiter	Atom
Atom	Atom

Exceptions

instantiation_error
The argument *Delimiter* or *List* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(atom_or_integer)
An element of *List* must be an atom, an integer or an integer expression, but is a term of another type.

type_error(atom)
The argument *Delimiter* must be an atom, but is a term of another type.

type_error(atom)
The argument *Atom* must be a variable or an atom, but is a term of another type.

Example

```
[user] ?- concat_atom([a,b,c],A).<ENTER>

A       = abc <ENTER>

yes
```

```
[user] ?- concat_atom([a,b,c],'_',B).<ENTER>

B       = a_b_c <ENTER>

yes

[user] ?- concat_atom([a,b,15],X).<ENTER>

X       = ab15 <ENTER>

yes

[user] ?- concat_atom([a,*],Y).<ENTER>

Y       = 'a*' <ENTER>

yes

[user] ?- concat_atom([@,*],Z).<ENTER>

Z       = @* <ENTER>

yes

[user] ?- concat_atom([p,a,n,g],pang).<ENTER>

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

append/3, atom_concat/3, write_atom/2, match_atom/2, atom_split/3

Connect a head and a tail to form a list

connect(?*List*, ?*Head*, ?*Tail* **)**

The predicate `connect/3` unifies the head of *List* with *Head* and the rest with *Tail*.

Arguments

List	List
Head	Term
Tail	Term

Hints

The call

```
connect(List, Head, Tail)
```

is identical to

```
List = [ Head | Tail ] .
```

Exceptions

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

Example

```
[user] ?- connect(L,alpha,[beta]).<ENTER>

L       = [alpha,beta] <ENTER>

yes

[user] ?- connect([a,b,c], Head, Tail ).<ENTER>

Head    = a
Tail    = [b,c] ;<ENTER>

no
```

The last element of *List* need not necessarily be the empty list.

```
[user] ?- connect([h | t],X,Y).<ENTER>

X       = h
Y       = t <ENTER>

yes
```

The call can also be made exclusively with uninstantiated variables.

```
[user] ?- connect(X,Y,Z), X = [Y|Z].<ENTER>

X       = [_69|_70]
Y       = _69
Z       = _70 <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

append/3

Read in a Prolog file

consult(+*Filename* **)** `[` **@** +*Module* `]`
`[` +*Filename* `]` `[` **@** +*Module* `]`

The predicate `consult/1` reads the file *Filename* into the database in the calling module or in the specified *Module*.

To enable Prolog text to be entered interactively as well, the atom `user` may also be specified for *Filename*. In this case, `consult/1` reads from the current input stream and displays a `'>'` prompt to indicate that a clause or a directive is expected and a `'|'` prompt to indicate that a clause or a directive that has already begun has not yet been completed. By entering `end_of_file` or pressing the appropriate key it is possible to terminate the consult process.

Clauses occurring in *Filename* are read in with `read_term/2`, checked for correct syntax and inserted with `assertz/1` or `assertz_with_names/2` into the database of the relevant module. Clauses with syntax errors are not inserted into the database.

If there are already clauses in the database for a predicate defined in *Filename*, they are not deleted before the new clauses are inserted in the database. However, this applies only if these clauses are not part of an explicitly defined module body in *Filename*, since the definition of a module body (see `begin_module/1`) will first delete any existing body of the same module.

If the Prolog flag `debug` is set to `on`, `assertz_with_names/2` is used for insertion in the database so that, for example, `listing/0/1` can be used to display the clauses of these predicates with their original variable names. Otherwise, `assertz/1` is used.

If grammar rules formulated with `-->/2` occur in *Filename*, they are interpreted by `consult/1` and converted into normal Prolog clauses.

Directives occurring in *Filename* are evaluated and executed according to their functionality. This may affect the further operation of `consult/1` (operator and module definitions for example). The following directives are available:

Directive	Description
`op/3`	Define and delete operators
`char_conversion/2`	Define a character conversion
`include/1`	Include a Prolog file in the source text
`ensure_loaded/1`	Insert a Prolog file in the source text once
`discontiguous/1`	Declare discontiguous predicate definition
`multifile/1`	Declare predicates as being scattered over several files
`init/1`	Initialization goal
`initialization/1`	Initialization goal
`dynamic/1`	Declare predicates as modifiable
`module/1`	Define a module
`begin_module/1`	Define the body of a module
`end_module/0/1`	Conclude the definition of a module

`import/1/2`	Import predicates
`export/1`	Export predicates of current module
`reexport/1`	Reexport predicates of imported module
`meta/1`	Declare metapredicates
`private/1`	Declare predicates as private

Arguments

Filename	Atom, name of a file or the atom `user`

Exceptions

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Filename*.

permission_error(modify,static_procedure)
When loading the file *Filename*, a predicate is to be defined which already exists and which may not be modified.

In addition, all the error messages and warnings from `read_term/2`, `assertz/1` and `assertz_with_names/2` and may occur. These do not cause the load process to be aborted. You should be aware that the database may not necessarily contain what you expect it to contain.

Errors in directives are also output as error messages on the error output stream. Unknown directives are reported by the error message `domain_error(directive)`.

In the event of errors in a module interface, the affected module is deleted after *Filename* has been processed.

If there are errors in the declaration part of a module body, the affected module body is deleted after *Filename* has been processed.

When initialization directives are executed (see `init/1`), any exception may arise. These aborts the execution of the initialization goal, but does not abort the load process.

In addition, warnings are output as a function of the Prolog flag `warnings` in certain situations to alert you to possible sources of errors.

Hints

If *Filename* does not contain the extension *.pro* and if a file of this name cannot be found, the extension *.pro* is appended and the resultant filename used.

The predicate `consult/1` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

If *Filename* contains module definitions, then database manipulation takes place in these modules. If not, it is implicitly assumed that the contents of *Filename* belong to the body of the calling module or of the specified *Module*. The difference compared with an explicitly specified `body/1` directive at the start of *Filename* is that the associated body of the module is not first deleted.

If *Filename* contains only module interfaces and/or bodies, then `consult/1` and `reconsult/1` are prefectly identical. The definition of a module interface of an existing module with the same name is first deleted, and the definition of a module body overwrites an existing module body.

The `init/1` directives in *Filename* are also executed in the context of the relevant module.

If a module in *Filename* contains several `init/1` directives, then the execution sequence is undefined.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

reconsult/1, ./2, `-->`/2

Create a context

context(+*Goal*, +*ContextInfo* **)** `[` **@** +*Module* `]`

The predicate `context/2` is used to create a context for executing *Goal*; the information specified in the list *ContextInfo* for context-global actions is stored in this context.

Three classes of context-global actions are supported:

1. **Context jumps**
   A context jump (similar to `longjmp()` in C) enables a context to be exited from any point in a search tree and the entire search tree activated by this context to be rejected at a stroke. When the context is exited, the actions defined can be executed.

   Context jumps can be initiated by the user (`throw/1`), or they can be executed automatically in the course of default error handling by IF/Prolog.

   If several contexts are nested, the **innermost** context **matching** the corresponding context jump is used (see section on *Context jumps* in *IF/Prolog User's Guide*[2]). If no matching context is found, IF/Prolog is exited with an error message.
   The predicate `break/0` (Prolog input loop) defines a context which **matches** all the context jumps initiated in the course of default error handling by IF/Prolog.

2. **Exceptions**
   Exceptions are raised by IF/Prolog if errors have occurred. However, they can also be raised explicitly by the user (`exception/2`). They are treated exactly as if they were exceptions raised by IF/Prolog.

   If there is information stored on how to handle the exception that has arisen for one of the active contexts (searching takes place from the inside to the outside), this exception handling is executed without the current search tree being discarded in the context. Exception handling **replaces** the goal which raised the exception. When it finishes, execution is continued normally if **no** context jump is executed during the handling.

   If an active context cannot be found which can handle the exception, a context jump is automatically initiated by IF/Prolog (see section on *Exceptions* in *IF/Prolog User's Guide*[2]).

3. **Signals**
   Signals are external events. One of the ways in which signals can be dealt with is active handling by the application (see section on *Signals* in *IF/Prolog User's Guide*[2]).

   If the application handles signals actively (see `signal_control/2`) and if there is information stored on how to handle the signal for one of the active contexts (searching takes place from the inside to the outside), the signal handling is executed without the current search tree being discarded, as in the case of context jumps.

   If handling is successful, the goal execution interrupted by the signal is resumed as if the signal had not been received. Otherwise, or if a context cannot be found which can handle the signal, the exception

```
error(system_error, [error=interrupt, signal=Signal, ...])
```

is created.

The *ContextInfo* list can contain a number of entries for each of the context-global actions. When an appropriate event occurs, the first set of information matching this event is used.

The following elements are valid in *ContextInfo*:

1. `catch(` *CatchMask* `) =>` *SubstituteGoal*
   If `throw/1` is called during execution within the context with a term which can be unified with *CatchMask*, the entire context is rejected and the `context/2` call replaced by *SubstituteGoal*.

2. `error(` *Error*`,` *Info* `) =>` *Handler*
   If an exception occurs within a context, or if `exception/2` is called, and the information belonging to the exception (see `exception/2`) can be unified with *Error* and *Info*, then the goal which raised the exception is replaced by a copy of *Handler* and execution is continued. The search tree in the context is not discarded.

3. `signal(` *Signal* `) =>` *Handler*
   If, during execution within the context, a signal occurs for which **active** handling is to take place (see `signal_control/2`) and its name can be unified with *Signal*, execution is interrupted, a copy of the goal *Handler* is called and, if this goal succeeds, the interrupted execution is resumed without the search tree being discarded in the context. If execution of *Handler* fails, the exception

   ```
   error(system_error, [error=interrupt, signal=Signal, goal=SubGoal])
   ```

   is created. If a predicate implemented in C is interrupted, the entry `interrupted=`*Predicate* is added to the above-mentioned list. In this case, *SubGoal* contains the goal that would follow the interrupted predicate.

Arguments

Goal	Goal
ContextInfo	List

Exceptions

instantiation_error
: The argument *ContextInfo* or *Goal* must not be a variable, but a variable was specified.

type_error(list)
: The argument *ContextInfo* must be a list, but is a term of another type.

type_error(context_indicator)
An element of *ContextInfo* must be a structure with the format *Mask* => *Handler*, but is a term of another type.

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate `context/2` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

The following predicate defines a daemon which becomes active when the signal `user_1` is received and, on completion of the activity, waits for further signals. The daemon terminates when the signal `user_2` is received.

```
daemon( WorkGoal ) :-
        signal_control( user_1, on ),
        signal_control( user_2, on ),
        context( sleep, [
                catch( end_of_work )
                        => true,
```

```
                    signal( user_1 )
                            => work( WorkGoal ),
                    signal( user_2 )
                            => throw( end_of_work )]).

sleep :-
        repeat,
        fail.

work( WorkGoal ) :-
        call( WorkGoal ),
        signal_control( user_1, on ).
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

call/1, catch/3, exception/2, throw/1, exception_handler/3, signal_handler/3, set_prolog_flag/2

Create a term copy

copy_term(@*Term1*, ?*Term2* **)**

The predicate `copy_term/2` unifies *Term2* with a copy of *Term1* in which all the variables have been replaced by new variables.

Arguments

Term1	Term
Term2	Term

Example

```
[user] ?- Orig = sni(Prolog),<ENTER>
          write(Orig),nl,<ENTER>
          copy_term(Orig,Copy),<ENTER>
          write(Copy),nl,<ENTER>
          Orig == Copy.<ENTER>
sni(_69)
sni(_226)

no
```

Identical variables in the term remain identical in the copy.

```
[user] ?- copy_term(f(A,A),X),<ENTER>
          X = f(Y,Z),<ENTER>
          Y == Z.<ENTER>

A       = _68
X       = f(_76,_76)
Y       = _76
Z       = _76 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

=/2, ==/2

Create an array

create_array(?*Array*, **@***Dimension*)

The predicate `create_array/2` creates an array of *Dimension* terms and unifies it with *Array*. Each element of the array is initialized to integer term 0 (zero).

Arguments

Array	Array (system-defined term)
Dimension	Positive integer

Exceptions

instantiation_error
The argument *Dimension* must not be a variable, but a variable was specified.

type_error(array)
The argument *Array* must be a variable or an array, but is a term of another type.

type_error(integer)
The argument *Dimension* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(greater_than_zero)
The argument *Dimension* must be an integer greater than zero, but is negative or equal to zero.

Furthermore, if an arithmetic expression is specified for the argument *Dimension*, all the exceptions for `is/2` can occur.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `create_array/2` is new.

See also

array/1/2, get_array/3, set_array/3

Query aliases of input/output streams

current_alias(?*Alias*, ?*Stream*)

The predicate `current_alias/2` queries the aliases of input/output streams.

If *Alias* is instantiated, *Stream* is unified with the input/output stream with which *Alias* is associated.

If *Stream* is instantiated, *Alias* is unified by backtracking with all the aliases associated with *Stream*.

If both are uninstantiated, *Alias* and *Stream* are unified by backtracking with all the alias associations known in the system.

The following aliases are built into the Prolog system:

Alias	Assignment
`user_input`	Standard input
`user_output`	Standard output
`user_error`	Standard error output
`keyboard`	Controlling terminal (input)
`screen`	Controlling terminal (output)
`current_input`	Current input
`current_output`	Current output
`current_error`	Current error output
`debug_input`	Used by the debugger
`debug_output`	Used by the debugger
`trace_output`	Used by the tracer

The predicate `current_alias/2` succeeds by backtracking for all the alias associations which were defined at the time of the first call, even if they have since been deleted or if other associations have been added (**logic update view**).

Arguments

Alias	Alias (atom)
Stream	Stream (system-defined term)

Exceptions

type_error(atom)
The argument *Alias* must be a variable or an atom, but is a term of another type.

domain_error(stream)
The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4`.

Hints

The sequence of solutions is undefined, if `current_alias/2` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

assign_alias/2, cancel_alias/1

Information on character conversions

current_char_conversion(?*CharacterIn*, ?*CharacterOut*)

The predicate `current_char_conversion/2` supplies information on the currently defined character conversions. Information is supplied only on conversions that do not map characters onto themselves.

During backtracking *CharacterIn* and *CharacterOut* are unified with a single character. This involves mapping the character *CharacterIn* to the character *CharacterOut* provided that this character is not enclosed in single or double quotes and is not used as a character constant.

The predicate `current_char_conversion/2` succeeds by backtracking for all *CharacterIn* and *CharacterOut* pairs which were defined at the time of the first call, even if the definitions have since been modified (**logic update view**).

Arguments

CharacterIn	Character
CharacterOut	Character

Exceptions

type_error(character)
The argument *CharacterIn* or *CharacterOut* must be a variable or a character, but is a term of another type.

Hints

The sequence of solutions is undefined, if `current_char_conversion/2` has several solutions for the current call.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

char_conversion/2

Query current module

current_default_module(?*Module*)

The predicate `current_default_module/1` unifies *Module* with the name of the current module.

Arguments

Module	Atom, name of a module

Exceptions

type_error(atom)
The argument *Module* must be a variable or an atom, but is a term of another type.

Hints

When IF/Prolog starts, the built-in module `user` is the current module.

In the input loop of IF/Prolog (`break/0`), the current module is always shown in the prompt.

All Prolog queries entered in the input loop of IF/Prolog (`break/0`) are executed in the current module unless the goal contains explicit module qualifications with `@/2`.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

@/2, module/1, set_default_module/1, current_module/1

Query information on device drivers

current_device_control(+*Device*, ?*Command*, ?*Arity* **)**

The predicate `current_device_control/3` unifies *Command* and *Arity* by backtracking with the information on the commands valid for *Device* to control stream-independent devices (see `device_control/2`).

This predicate provides a means of checking whether a device driver provides certain stream-independent functionality.

Since IF/Prolog can be equipped with any device drivers using its C driver interface, it is always best before using `device_control/2` to check whether the driver can in fact execute it.

Arguments

Device	Atom, device driver name
Command	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Device* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Device* must be an atom, but is a term of another type.

domain_error(device_name)
The value of the argument *Device* is not an existing device driver.

type_error(atom)
The argument *Command* must be a variable or an atom, but is a term of another type.

type_error(integer)
The argument *Arity* must be a variable or an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Arity* must be a positive integer, but is negative.

representation_error(max_arity)
The argument *Arity* must be an integer within the range 0..127.

Hints

The sequence of solutions is undefined, if `current_device_control/3` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

device_control/2, stream_device/2, current_stream_control/3

Query current error output stream

current_error(?*Stream*)

The predicate `current_error/1` unifies *Stream* with the current error output stream.

Arguments

Stream	Stream (system-defined term)

Exceptions

domain_error(stream)
The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4`.

Hints

The predicate `current_error/1` will never unify *Stream* with an alias or the filename that was possibly used with `error_tell/1` or `open/4`.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_input/1, current_ouput/1, error_telling/1, set_input/1, set_output/1, set_error/1

Query global variables

current_global(?*Name*) [@ +*Module*]

The predicate `current_global/1` unifies *Name* by backtracking with all the global variables defined in the calling module or in the specified *Module*.

The predicate `current_global/1` succeeds by backtracking for all the global variables which were defined at the time of the first call, even if they have since been deleted or if other associations have been added (**logic update view**).

Arguments

Name	Atom

Exceptions

type_error(atom)
The argument *Name* must be a variable or an atom, but is a term of another type.

Hints

The sequence of solutions is undefined, if `current_global/1` has several solutions for the current call.

Global variables are always assigned to a module; they can be neither exported nor imported. In operations with global variables, therefore, only those global variables defined in the calling module or in the specified *Module* are taken into account.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

get_global/2, pop_global/2, push_global/2, set_global/2, unset_global/1

Query names of current host

current_host(?*Host*)

The predicate `current_host/1` queries the name(s) of the current host.

The argument *Host* is unified by backtracking with the standard name and the alias names of the current host. The names are obtained either from the name server (`named`), the NIS service or the `hosts` database (on UNIX: */etc/hosts* or */etc/inet/hosts*).

Arguments

Host	Atom

Exceptions

type_error(atom)
The argument *Host* must be a variable or an atom, but is a term of another type.

Hints

The sequence of solutions is undefined, if `current_host/1` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

host_addr/2, net_service/3, socket/3

Query current input stream

current_input(?*Stream*)

The predicate `current_input/1` unifies *Stream* with the current input stream.

Arguments

Stream	Stream (system-defined term)

Exceptions

domain_error(stream)
The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4`.

Hints

The predicate `current_input/1` will never unify *Stream* with an alias or the filename that was possibly used with `see/1` or `open/4`.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

current_output/1, current_error/1, seeing/1, set_input/1, set_output/1, set_error/1

Query current language

current_language(?*Language*)

The predicate `current_language/1` unifies *Language* with the current language.

The language can be set with the command line argument `-language`. It specifies, which character set is supported for text streams.

If IF/Prolog has been configured with multiple-octet characters, following language specifications are supported:

ascii : ASCII character set (**default**)

euc : EUC character set

sjis : SJIS character set

chinese : Chinese character set

korean : Korean character set

taiwanese : Taiwanese character set

If IF/Prolog has not been configured with multiple-octet characters, only the character set **ascii** is available.

Arguments

Language	Atom

Exceptions

type_error(atom)
The argument *Language* must be a variable or an atom, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Query memory management parameters

current_memory_management(?*Area*, ?*Parameter*, ?*Value*)

The predicate `current_memory_management/3` unifies *Area*, *Parameter* and *Value* by backtracking with the values assigned to one of the memory management parameters. IF/Prolog manages the memory areas `database`, `global`, `constraints`, `local`, `trail` and `extension`. Each possesses the parameters `minimal_size`, `maximal_size`, `minimal_gap`, `maximal_gap`, `default_gap` and `trace`. The values of all but the last parameter are positive integers. The parameter `trace` has the value `on` or `off`.

Arguments

Area	Atom: database | global | constraints | local | trail | extension
Parameter	Atom: minimal_size | maximal_size | minimal_gap | maximal_gap | default_gap | trace
Value	Integer (*Value* ≥ 0) or atom (*Value* == on | off)

Exceptions

type_error(atom)
: The argument *Area*, *Parameter* or *Value* must be a variable or an atom, but is a term of another type.

type_error(integer)
: The argument *Value* must be a variable or an integer, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

set_memory_management/3, garbage_collection/1, statistics/0/2

Query modules

current_module(?*Module*)

The predicate `current_module/1` unifies *Module* by backtracking with all the module names known to IF/Prolog.

There are a number of built-in module names in IF/Prolog which cannot be redefined by the user:

Module	Meaning
`system`	Built-in predicates
`system...`	Further system modules
`motif...`	Motif interface
`informix...`	Informix interface
`const_...`	Constraint package modules

Further names for additional interfaces of IF/Prolog may be added.

The predicate `current_module/1` succeeds by backtracking for all the modules which existed at the time of the first call, even if they have since been deleted or others have been added (**logic update view**).

Arguments

Module	Atom, name of a module

Exceptions

type_error(atom)
: The argument *Module* must be a variable or an atom, but is a term of another type.

Hints

A module name is still known even if its interface and its body have been deleted by `unload/1`.

The sequence of solutions is undefined, if `current_module/1` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, set_default_module/2, current_default_module/1

Information on operators

current_op(?*Priority*, ?*Assoc*, ?*Name*)

The predicate `current_op/3` supplies information on the operators defined.

During backtracking, *Name* is unified with the name of the operator, *Assoc* with the identifier for its associativity (see `op/3`) and *Priority* with its operator precedence.

The predicate `current_op/3` succeeds by backtracking for all the operators which existed at the time of the first call, even if they have since been deleted or others have been added (**logic update view**).

Arguments

Priority	Integer, 1..1200
Assoc	Atom: xf | yf | fx | fy | xfx | xfy | yfx
Name	Atom

Exceptions

type_error(integer)
: The argument *Priority* must be a variable or an integer, but is a term of another type.

type_error(atom)
: The argument *Assoc* or *Name* must be a variable or an atom, but is a term of another type.

Hints

The sequence of solutions is undefined, if `current_op/3` has several solutions for the current call.

Operators are not associated with a module, but are globally visible.

Example

Output of all operators.

```
[user] ?- current_op(V,T,N),<ENTER>
          writeq(op(V,T,N)),nl,fail.<ENTER>
op(1000,xfy,',')
op(50,xfx,:)
op(100,xfx,@)
op(200,fy,\)
op(200,xfy,^)
op(200,xfx,**)
op(400,yfx,rdiv)
op(400,yfx,>>)
op(400,yfx,<<)
op(400,yfx,rem)
op(400,yfx,mod)
op(400,yfx,*)
op(400,yfx,/)
op(400,yfx,//)
op(500,yfx,/\)
op(500,yfx,\/)
op(200,fy,-)
op(500,yfx,-)
op(200,fy,+)
op(500,yfx,+)
op(700,xfx,=)
op(700,xfx,=..)
op(700,xfx,<)
op(700,xfx,=<)
op(700,xfx,>)
op(700,xfx,>=)
op(700,xfx,=:=)
op(700,xfx,==)
op(700,xfx,=\=)
op(700,xfx,@<)
op(700,xfx,@=<)
op(700,xfx,@>)
op(700,xfx,@>=)
op(700,xfx,@\=)
op(700,xfx,@=)
op(700,xfx,\=)
op(700,xfx,\==)
op(700,xfx,is)
op(900,xfx,=>)
op(900,fy,\+)
op(900,fy,not)
op(1050,xfy,->)
op(1100,xfy,;)
```

```
op(1100,xfx,from)
op(1150,fx,reexport)
op(1150,fx,export)
op(1150,fx,import)
op(1150,fx,meta)
op(1150,fx,end_module)
op(1150,fx,begin_module)
op(1150,fx,module)
op(1150,fx,ensure_loaded)
op(1150,fx,include)
op(1150,fx,initialization)
op(1150,fx,init)
op(1150,fx,private)
op(1150,fx,dynamic)
op(1150,fx,multifile)
op(1150,fx,discontiguous)
op(1200,xfx,-->)
op(1200,xfx,:-)
op(1200,fx,:-)
op(1200,fx,?-)

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

op/3

Query current output stream

current_output(?*Stream*)

The predicate `current_output/1` unifies *Stream* with the current output stream.

Arguments

Stream	Stream (system-defined term)

Exceptions

domain_error(stream)
The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4`.

Hints

The predicate `current_output/1` will never unify *Stream* with an alias or the filename that was possibly used with `tell/1` or `open/4`.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

current_input/1, current_error/1, telling/1, set_input/1, set_output/1, set_error/1

Query visible predicates

current_predicate(?*Predicate* **) [@** +*Module* **]**

The predicate `current_predicate/1` unifies *Predicate* by backtracking with the predicate indicators of all the visible predicates in the calling module or in the specified *Module*.

The predicate `current_predicate/1` succeeds by backtracking for all the predicates which were visible at the time of the first call, even if they have since been deleted or if other definitions have been added (**logic update view**).

Arguments

Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

permission_error(access,system_module)
The predicate `current_predicate/1` must not be executed for a system module.

type_error(predicate_indicator)
The argument *Predicate* must be a variable or a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

Hints

The predicate `current_predicate/1` is a metapredicate and considers only predicate definitions which are visible in the calling module or in the specified *Module*.

The sequence of solutions is undefined, if `current_predicate/1` has several solutions for the current call.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

current_visible/2, predicate_type/2

Query Prolog flags

current_prolog_flag(?*Flag*, ?*Value*)

The predicate `current_prolog_flag/2` unifies *Flag* by backtracking with all the names of Prolog flags known in IF/Prolog and *Value* with its current value.

Prolog flags are used to make global settings for IF/Prolog.

The following tables contain the available Prolog flags and their permissible values (default values printed in **boldface**).

The following Prolog flags may be queried and modified:

Flag	Value	Description
`consult`	**notify**	A message is output when a Prolog text is loaded.
	`nonotify`	No message is output when a Prolog text is loaded.
`search_path`	*Paths*	If input files are not specified using a complete path name, they are searched for in the directories specified in the *Paths* list. The default value of this Prolog flag is the value of the system parameter `-sp`, if specified, otherwise it is the value of the environment variable `PROPATH` or it is empty.
`read_error`	**error**	Syntax errors found by analysis with `read_term/2/3`, etc. raise an exception.
	`fail`	Syntax errors found by analysis with `read_term/2/3`, etc. are reported directly by the parser in a message and the predicate fails.
	`quiet`	Syntax errors found by analysis with `read_term/2/3`, etc. are not reported and the predicate fails.
`report_read_error`		
	on	Syntax errors found by analysis with `parse_atom/6` are reported directly by the parser in a message.
	`off`	Syntax errors found by analysis with `parse_atom/6` are not reported by the parser but can be queried with the `read_error/2/3` predicates.
`warnings`	**on**	For `consult/1`, etc. warnings are output for possible errors.
	`off`	For `consult/1`, etc. no warnings are output.

`unknown`	**error**	If a predicate is to be activated which is not defined, an exception is raised. This does not apply to predicates for which the exception mode has been set to `on` (see `predicate_mode/3`).
	`fail`	If a predicate is to be activated which is not defined, backtracking is initiated, i.e. the predicate is evaluated with `fail`.
	`warning`	If a predicate is to be activated which is not defined, a warning is output and backtracking is initiated, i.e. this predicate is evaluated with `fail`. The warning is not output for predicates for which the exception mode has been set to `on` (see `predicate_mode/3`).
`debug`	`on`	For `consult/1` and `reconsult/1`, *all* predicates are implicitly declared as dynamic (see `dynamic/1`) to support program testing.
	off	For `consult/1` and `reconsult/1`, all the predicates which were not explicitly declared with `dynamic/1` are declared as static. These cannot then be modified, tested or displayed.
`prompt`	`on`	Output of the prompt in the Prolog input loop and interaction for backtracking for goals which have been successfully executed in the Prolog input loop and for which variables have been instantiated *always* take place, i.e. even if the standard input and the standard output are not terminals.
	off	Output of the prompt in the Prolog input loop and interaction for backtracking for goals which have been successfully executed in the Prolog input loop and for which variables have been instantiated take place *only* if the standard input and the standard output are terminals.
`char_conversion`	**on**	The character conversions defined with the predicate `char_conversion/2` are performed when terms are read.
	`off`	No character conversion is performed when terms are read.

`backtrace`	**10**	The maximal length of the **backtrace**-list. The list is created when an exception has been raised and contains the calling sequence of goals that has led to the exception. The elements of the list are predicate indicators (*Functor*/*Arity*). Goals that have been removed due to tail-recursion optimization (normally the last subgoal in a clause body) are not included in the list.
`double_quotes`	**codes**	Characters inside double quotes (") are to be interpreted as a list of character codes when reading a term (e.g. with `read/1/2`).
	`chars`	Characters inside double quotes (") are to be interpreted as a list of characters when reading a term (e.g. with `read/1/2`).
	`atom`	Characters inside double quotes (") are to be interpreted as an atom when reading a term (e.g. with `read/1/2`).
`extended_syntax`	`on`	Cyclic terms can be entered using the notation *Variable* `::` *Term*. When reading such a cyclic term, *Variable* is unified with *Term*. The above notation is also used by output of cyclic terms (see `=/2`).
	off	Cyclic terms can be entered in the above notation, but *Variable* is **not** unified with *Term*.
`nested_comments`	**off**	Nested comments are not allowed in a Prolog text. Begin of a comment inside a comment is ignored, which normally leads to a syntax error.
	`on`	Nested comments are accepted in a Prolog text.
`write_depth`	**10**	Maximal depth of terms output in the Prolog toplevel (see `break/0`). This applies to variable instantiations of a query and to exception messages. The predicate `write_formatted/2/3` takes this value in account, if a term is to be output with the format `%w`.

The following Prolog flags can only be queried but not changed. Their value depends on the command line parameters specified when the Prolog system was started (see `system_parameters/1`).

Flag	Value	Description
`signal`	`on`	The system parameter `-nosignal` was **not** specified.
	`off`	The system parameter `-nosignal` was specified.

`notty`	`on`	The system parameter `-notty` was specified.
	`off`	The system parameter `-notty` was **not** specified.
`iso`	`on`	The system parameter `-iso` was specified.
	`off`	The system parameter `-iso` was **not** specified.

The following Prolog flags supply information on the implementation of the Prolog system or on the operating system (the values cannot be changed):

Flag	Description
`max_arity`	The value indicates the maximum arity of predicates and structures.
`min_integer`	The value indicates the smallest representable machine-precision (32- or 64-bit) integer.
`max_integer`	The value indicates the largest representable machine-precision (32- or 64-bit) integer.
`bounded`	This flag can assume the following values: `true` Integer arithmetic returns correct values only if the operands and the mathematically correct result lie within the closed interval (`min_integer`, `max_integer`). `false` Integer arithmetic always returns correct results or an exception is created if this is not possible.
`integer_rounding_function`	This flag can assume the following values: `down` The arithmetic functions `//` and `rem` round their result down if necessary. `toward_zero` The arithmetic functions `//` and `rem` round the absolute value of their result down if necessary.
`float_min`	The value indicates the smallest representable floating-point number.
`float_max`	The value indicates the largest representable floating-point number.

The predicate `current_prolog_flag/2` succeeds by backtracking for all the Prolog flags and values which were set at the time of the first call, even if they have since been modified (**logic update view**).

Arguments

Flag	Atom
Value	Term

Exceptions

type_error(atom)
The argument *Flag* must be a variable or an atom, but is a term of another type.

domain_error(prolog_flag)
The value of the argument *Flag* is not a valid Prolog flag.

Hints

The default settings for the Prolog flags `consult`, `debug`, `prompt`, `search_path` and `warnings` can be modified by system parameters (see `system_parameters/1`) when IF/Prolog is started.

The values of the Prolog flags `notty`, `iso` and `signal` can be set using the appropriate system parameters **only** when IF/Prolog is started.

Example

```
[user] ?- current_prolog_flag(F,W),<ENTER>
          write(F),write(' = '),<ENTER>
          write(W),nl,fail.<ENTER>
bounded = false
char_conversion = on
consult = notify
debug = off
float_max = 1.79769e+308
float_min = 4.94066e-324
integer_rounding_function = toward_zero
iso = off
max_arity = 127
max_integer = 2147483647
min_integer = -2147483648
notty = off
prompt = off
read_error = error
report_read_error = on
signal = on
search_path = []
undefined_predicate = error
warnings = on

no

[user] ?- current_prolog_flag(consult, _).<ENTER>

yes

[user] ?- current_prolog_flag(warnings, on).<ENTER>
```

```
yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The Prolog flags `backtrace`, `consult`, `double_quotes`, `extended_syntax`, `float_max`, `float_min`, `iso`, `nested_comments`, `notty`, `prompt`, `read_error`, `report_read_error`, `search_path`, `signal`, `warnings` and `write_depth` are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The value of the Prolog flags `bounded`, `max_integer` and `min_integer` depends on whether **Multiple Precision Integer and Rational Number Support** has been configured.

See also

set_prolog_flag/2, system_parameters/1

Query signals and modes

current_signal(?*Signal*, ?*Mode*)

The predicate `current_signal/2` unifies *Signal* by backtracking with all the signal names known in IF/Prolog and *Mode* with their current processing mode.

The following signal names are known:

Prolog name	C name
`interrupt`	SIGINT
`abort`	SIGABRT
`alarm`	SIGALRM
`pipe`	SIGPIPE
`quit`	SIGQUIT
`termination`	SIGTERM
`user_1`	SIGUSR1
`user_2`	SIGUSR2

The processing modes for signals are `on`, `off`, `default` or `ignore` (see `signal_control/2`).

The predicate `current_signal/2` succeeds by backtracking for all the signals and modes which were set at the time of the first call, even if they have since been modified (**logic update view**).

Arguments

Signal	Atom: abort | alarm | interrupt | pipe | quit | termination | user_1 | user_2
Mode	Atom: on | off | default | ignore

Exceptions

type_error(atom)
The argument *Signal* or *Mode* must be a variable or an atom, but is a term of another type.

Hints

The sequence of solutions is undefined, if `current_signal/2` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

signal_handler/3, signal_control/2, context/2

Query communication sockets

current_socket(?*Domain*, ?*Type*, ?*Socket*)

The predicate `current_socket/3` queries endpoints for communication.

The argument *Domain* is unified with the communications domain within which communication takes place.

The argument *Type* is unified with the semantics of the communication.

The argument *Socket* is unified by backtracking with a system-generated term.

Arguments

Domain	Atom: internet | unix
Type	Atom: datagram | stream
Socket	Socket (system-defined term)

Exceptions

type_error(atom)
: The argument *Domain* or *Type* must be a variable or an atom, but is a term of another type.

type_error(socket)
: The argument *Socket* must be a variable or a socket supplied by `socket/3`, but is a term of another type.

Hints

The sequence of solutions is undefined, if `current_socket/3` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

socket/3, get_socket_option/3, set_socket_option/3, net_service/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Query information on device drivers

current_stream_control(@*Stream*, ?*Command*, ?*Arity*)

The predicate `current_stream_control/3` unifies *Command* and *Arity* by backtracking with information on the commands valid for *Stream* to control input/output streams (see `stream_control/2`).

This predicate provides a means of checking whether the device driver of an input/output stream implements certain functionalities.

For *Stream* the name of a device driver may be specified (see `stream_device/2`) so that checks can be carried out before a stream is opened.

Since IF/Prolog can be equipped with any device drivers using its C driver interface, it is always best to check whether the driver of the current stream can in fact execute a specific command before using `stream_control/2`.

Arguments

Stream	Stream (system-defined term), alias (atom) or device driver name
Command	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

type_error(atom)
The argument *Command* must be a variable or an atom, but is a term of another type.

type_error(integer)
The argument *Arity* must be a variable or an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Arity* must be a positive integer, but is negative.

Hints

The sequence of solutions is undefined, if `current_stream_control/3` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

stream_control/2, stream_device/2, device_control/2, current_device_control/3

Query visible predicates

current_visible(?*DefModule*, ?*Predicate*) [@ +*Module*]

The predicate `current_visible/2` unifies *Predicate* by backtracking with the predicate indicators of all the *visible* predicates in the calling module or in the specified *Module* and *DefModule* with the name of the module in which *Predicate* was defined.

A predicate is regarded as visible in a module if it was defined or declared in this module or if it was imported from another module.

The predicate `current_visible/2` succeeds by backtracking for all the predicates which were visible at the time of the first call, even if they have since been deleted or if other associations have been added (**logic update view**).

Arguments

DefModule	Atom, name of a module
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

permission_error(access,system_module)
The predicate `current_visible/2` must not be executed for a system module.

type_error(atom)
The argument *DefModule* must be a variable or an atom, but is a term of another type.

type_error(predicate_indicator)
The argument *Predicate* must be a variable or a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

Hints

The sequence of solutions is undefined, if `current_visible/2` has several solutions for the current call.

The built-in predicates are visible in each module without having to be explicitly imported.

The language constructs (such as `->/2`) are not predicates and are therefore not supplied by `current_visible/2`.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_predicate/1

Test for cyclic term

cyclic(@TestTerm)

The predicate `cyclic/1` succeeds if *TestTerm* is a cyclic term, otherwise it fails. Cyclic terms can be created with unification (e.g. with the predicate `=/2`).

Arguments

TestTerm	Term

Example

```
[user] ?- cyclic(a(A,A)).<ENTER>

no

[user] ?- A = s(A), cyclic(A).<ENTER>

A       = _155 :: s(_155) <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `cyclic/1` is new.

See also

=/2, unify_with_occurs_check/2, atom/1, atomic/1, compound/1, float/1, ground/1, integer/1, nonvar/1, number/1, rational/1, var/1

Activate the debugger for a goal

debug(+*Goal* **)** `[` **@** +*Module* `]`

The predicate `debug/1` activates the debugger for executing *Goal*.

Before *Goal* is executed, the user-defined predicate `debug_begin/0` is called, if it is visible in the calling module or in the specified *Module*.

After *Goal* has been executed, the user-defined predicate `debug_end/0` is called, if it is visible in the calling module or in the specified *Module*.

Arguments

Goal	Goal

Exceptions

instantiation_error
: The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
: The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
: In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
: The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
: The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

If `debug/1` is called within `trace/1` or `debug/1`, this call is ignored.

The predicate `debug/1` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in

the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

call/1, debug_begin/0, debug_end/0, trace/1, set_prolog_flag/2

User-defined debugger initialization

debug_begin [:- *Body*]

The predicate `debug_begin/0` is not a built-in predicate but one which can be defined by the user. It is called in the module in which `debug/1` was called.

It is called automatically by `debug/1` when the debugger is started.

The user can define `debug_begin/0` to perform preliminary activities, such as defining default breakpoints with `debug_config/3`, when the debugger is started.

Hints

Nested calls involving `debug/1` and/or `trace/1` are ignored. If `debug/1` is called several times during a goal execution, `debug_begin/0` is called each time.

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

debug/1, debug_end/0, debug_config/3

Configure the debugger

debug_config(+*Parameter*, ?*Old*, +*New* **)**

The predicate `debug_config/3` is used to query and modify debugger parameters.

Parameter is used to specify the debugger parameter to be queried or modified. At the time of the call, *Old* is unified with the previous value of the debugger parameter and then, if unification was successful, the new value *New* is set. The following debugger parameters are available (default values are printed in boldface):

Parameter	Value	Description
`call`	**on**	The interactive debugger stops at each `CALL` port (implicit breakpoint).
	`off`	The interactive debugger will not normally stop at the `CALL` port unless a user-defined breakpoint exists or an execute command requires it to.
`exit`	**on**/`off`	the same applies to the `EXIT` port
`redo`	**on**/`off`	the same applies to the `REDO` port
`fail`	**on**/`off`	the same applies to the `FAIL` port
`error`	**on**/`off`	the same applies to the `ERROR` port
`wakeup`	**on**/`off`	the same applies to the `WAKEUP` port
`suspend`	**on**/`off`	the same applies to the `SUSPEND` port
`trymatch`	`on`/**off**	the same applies to the `TRYMATCH` port
`failmatch`	`on`/**off**	the same applies to the `FAILMATCH` port
`enterbody`	`on`/**off**	the same applies to the `ENTERBODY` port
`exitbody`	`on`/**off**	the same applies to the `EXITBODY` port
`redobody`	`on`/**off**	the same applies to the `REDOBODY` port
`failbody`	`on`/**off**	the same applies to the `FAILBODY` port
`display_depth`	**7**	The maximum structure depth is restricted by this parameter when terms are output by the debugger. If `0` is specified, there is no restriction.
`history`	**3**	The number of goal and clause boxes to be stored for the alpha and Motif interfaces.
`trace`	`on`	Tracing in activated.
	off	Tracing is deactivated.

`display`	**automatic**	The user interface is set automatically by the system in accordance with the system environment. This value is the default setting unless the system parameter `stream` was specified when the Prolog system was started (see `system_parameters/1`).
	`alpha`	The most basic interface for alphanumeric screens is set. The I/O device must be a terminal and the device driver must be able to perform terminal control functions. If this is not the case, the value is automatically reset to `stream`.
	`stream`	The most basic command-oriented interface is set. It does not require any terminal functions, i.e. it can also be used for redirected input/output. This value is the default setting if the system parameter `stream` was specified when the Prolog system was started (see `system_parameters/1`).
	`motif`	An attempt is made to start the debugger's Motif interface on the default X display (see Motif documentation). If this attempt is unsuccessful, the value is automatically reset to `alpha`.
	`motif(`*Display*`)`	*Display* is regarded as the specification of an X display, and an attempt is made to start the debugger's Motif interface on this X display. If this attempt is not successful, the value is automatically reset to `alpha`.

Arguments

Parameter	Atom, see table above
Old	Term, context-dependent
New	Term, context-dependent

Exceptions

instantiation_error
The argument *Parameter* or *New* or a subterm must not be a variable, but a variable was specified.

type_error(atom)
The argument *Parameter* must be an atom, but is a term of another type.

domain_error(config_option)
The argument *Parameter* is not a valid configuration parameter.

type_error(integer)
The argument *New* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *New* must be a positive integer, but is negative.

type_error(atom)
The argument *New* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *New* must be one of the atoms `on` or `off`, but is another atom.

domain_error(display_option)
The value of the argument *New* is not a valid display specification.

Furthermore, if an arithmetic expression is specified for the argument *New*, all the exceptions for `is/2` can occur.

Hints

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

debug/1, trace_config/3, system_parameters/1

User-defined termination of the debugger

debug_end `[ :-` *Body* `]`

The predicate `debug_end/0` is not a built-in predicate but one which can be defined by the user. It is called in the module in which `debug/1` was called.

It is called automatically after `debug/1` has been processed.

The user can define `debug_end/0` to perform terminating tasks, such as closing files, when the debugger is deactivated.

Hints

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

debug/1, debug_begin/0

Query and set the debug mode of a predicate

debug_mode(@*Predicate*, ?*Before*, +*After* **)** [@ +*Module*]

The predicate `debug_mode/3` defines, whether the *Predicate* is to be monitored by the debugger or the tracer of IF/Prolog.

Identification of the predicate is based on its visibility in the calling module or in the specified *Module*.

The argument *Before* is unified with the previously valid mode. The debug mode is then set according to *After*.

The debug mode can only be changed for predicates for which the modification mode is set to `on`. The built-in predicates, like C predicates, can never be monitored.

If *After* is

`on` then the predicate *Predicate* can be monitored, i.e. subgoals of the predicate can also be investigated.

`off` then *Predicate* cannot be monitored.

The predicate *Predicate* need not yet be defined. This attribute is retained beyond `abolish/1`.

Arguments

Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$
Before	Atom: on | off
After	Atom: on | off

Exceptions

instantiation_error
The argument *Predicate* or *After* or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
The argument *Predicate* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

type_error(atom)
The argument *After* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *After* must be one of the atoms `on` or `off`, but is another atom.

type_error(atom)
The argument *Before* must be a variable or an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Before* must be a variable or one of the atoms `on` or `off`, but is another atom.

permission_error(access,system_module)
The predicate `debug_mode/3` must not be executed for a system module.

Hints

The predicate `debug_mode/3` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

A predicate may have certain attributes, which, when it is created, are set implicitly by the system according to the current environment or declarations. They can be modified to a certain extent explicitly by the user. These include the following:

- List mode (see `list_mode/3`)
- Debug mode (see `debug_mode/3`)
- Exception mode (see `predicate_mode/3`)
- Modification mode (see `modify_mode/3`)

These attributes are retained even if the predicate is deleted (with `abolish/1` for example).

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

abolish/1, debug/0, trace/0, dynamic/1, modify_mode/3

Query and set the editor

default_editor(?*EditorOld*, +*EditorNew*)

The predicate `default_editor/2` unifies *EditorOld* with the name of the editor currently. If unification succeeds, the setting is changed to *EditorNew*.

The editor is used by `edit/0/1`.

Arguments

EditorOld	Atom
EditorNew	Atom

Exceptions

instantiation_error
The argument *EditorNew* must not be a variable, but a variable was specified.

type_error(atom)
The argument *EditorNew* must be an atom, but is a term of another type.

type_error(atom)
The argument *EditorOld* must be a variable or an atom, but is a term of another type.

Hints

With the environment variable `EDITOR` it is possible to determine which editor is set at system startup. If the variable is not set, `vi` is set.

The setting is changed to the required value, irrespective of whether the specified name represents a usable editor or not.

Example

Query the current editor:

```
[user] ?- default_editor(E, E).<ENTER>

E       = vi <ENTER>

yes
```

Set a new editor:

```
[user] ?- default_editor(_, emacs),<ENTER>
          default_editor(E, E).<ENTER>

E       = emacs <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

edit/0/1

Control an input/output device

device_control(+*Device*, @*Command* **)**

The predicate `device_control/2` enables the stream-independent control of input/output devices.

A device driver can provide a number of commands for executing specific operations such as configuration.

New device drivers can be linked into IF/Prolog using the C driver interface. The built-in device drivers of IF/Prolog implement no stream-independent control commands.

Arguments

Device	Atom, name of a device driver
Command	Term, device-dependent

Exceptions

instantiation_error
The argument *Device* or *Command* or a subterm must not be a variable, but a variable was specified.

type_error(atom)
The argument *Device* must be an atom, but is a term of another type.

domain_error(device_name)
The value of the argument *Device* is not an existing device driver.

type_error(atom_or_compound)
The argument *Command* must be an atom or a structure, but is a term of another type.

domain_error(device_control)
The value of the argument *Command* is none of the values permitted for a device control command.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_device_control/3, stream_device/2

Test for digit

digit(@*TestTerm* **)**

The predicate `digit/1` succeeds if *TestTerm* is an atom consisting of a digit character, otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- digit('7').<ENTER>

yes

[user] ?- digit(7).<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

letter/1, number/1, number_chars/2

Declare discontiguous predicate definition

:- discontiguous @*PredicateList*
:- discontiguous(@*PredicateList* **)**

A predicate declared with the directive `discontiguous/1` does not have to be defined textually in a single block, i.e. its definition can be interrupted by the definition of other predicates.

The directive `discontiguous/1` may only occur in the declaration part of a module body. The first `discontiguous/1` directive indicating a predicate must precede all clauses for this predicate.

Arguments

PredicateList	List with the format `[`*Predicate*,...`]` or the structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
: The argument *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(list)
: The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
: The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
: The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
: The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
: The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

permission_error(directive,module)
The specified directive `discontiguous/1` is not valid in this position.

Hints

The atom `discontiguous` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `discontiguous/1` complies with the definition of the ISO standard for Prolog.

See also

consult/1, reconsult/1, compile/1, load/1, multifile/1

Declare predicates as modifiable

:- dynamic @*PredicateList*
:- dynamic(@*PredicateList* **)**

Predicates declared with the directive `dynamic/1` can be modified after definition even if the module to which they belong is compiled or the Prolog flag `debug` is set to `off`.

All predicate attributes (list mode, debug mode, modification mode and exception mode) for the predicates listed in *PredicateList* are set to `on`.

The directive `dynamic/1` may only occur in the declaration part of a module body. The first `dynamic/1` directive indicating a predicate must precede all clauses for this predicate.

Arguments

PredicateList	List with the format `[`*Predicate*,...`]` or structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
: The argument *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(list)
: The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
: The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
: The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
: The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

permission_error(directive,module)
The specified directive `dynamic/1` is not valid in this position.

Hints

The atom `dynamic` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `dynamic/1` complies with the definition of the ISO standard for Prolog.

See also

compile/1, load/1, predicate_mode/3, modify_mode/3

Edit a file

edit [@ +*Module*]

edit(+*Filename* **)** [@ +*Module*]

The predicate `edit/1` calls an editor for the specified file *Filename*.

If *Filename* does not contain the extension *.pro* and if a file of this name cannot be opened for reading, the extension is appended and the resultant filename is used.

The predicate `edit/0` acts in the same way as `edit/1`, but edits the file last edited with `edit/1` without its name having to be specified again. The predicate `edit/0` fails, if `edit/1` has not been called since IF/Prolog started.

At the end of the editing operation you are asked:

`Should the file` *Filename* `be reconsulted (y/n)?`

If you answer `y` or `Y` (without blanks or other characters), the file *Filename* is automatically reconsulted (`reconsult/1`) in the calling module or in the specified *Module*. If any other input is made, the file will not be reconsulted. In either case, `edit/0/1` succeeds.

If the Prolog flag `debug` has the value `on`, the errors encountered during `reconsult/1` are collected. If `edit/0/1` is called again for this file, a temporary file is created containing the errors dispersed in the source code. The error messages are marked with '`>>>>`' at the beginning of each line. If the column of the error is also known, it is marked with '`^`'. After the temporary file has been edited (to correct the errors), the error information is removed and the temporary file is stored in *Filename*.

The predicates `edit/0/1` do not check the existence of the file to be edited. Behavior (e.g. if the file is created) then depends on the relevant editor.

The editor can be set with `default_editor/2`. The editor command is executed with `system/1`.

Most screen editors will only work well if the standard Prolog streams (`user_input`, `user_output` and `user_error`) are in fact connected to a terminal.

Arguments

Filename	Atom

Exceptions

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

system_error(command_failed)
A command passed to the operating system or a corresponding operating system function could not be executed successfully.

In addition, all the exceptions and error messages described for `reconsult/1` may occur.

Example

`edit/0/1` cannot determine whether the editor set is in fact a command deserving the name:

```
[user] ?- default_editor(_,cat).<ENTER>

yes

[user] ?- edit(example).<ENTER>
a :- b.
b :- true.
Should the file example.pro be reconsulted (y/n)? y<ENTER>
*** reconsult 'example.pro': loaded in 0.02 sec.

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

default_editor/2

Conclude the definition of a module

:- end_module

:- end_module +*Module*
:- end_module(+*Module* **)**

The directive `end_module/0/1` marks the end of the definition of the current module or the module *Module*. The directive `end_module/0/1` is used to complete as well the module interface as the module body.

The directive `end_module/1` can only be used to conclude the definition of the current module. This is why *Module* must be identical to the name of the current module.

The directive `end_module/0` concludes the definition of the current module. This assumes, of course, that a module is being defined when this directive is processed.

Once processing of the `end_module/0/1` directive has been completed, the system is located in the module `user`, i.e. all subsequent predicate definitions and directives relate to the module `user` until a new module is defined or until the load process is terminated.

Arguments

Module	Atom, name of a module

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *Module* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

existence_error(module)
The argument *Module* is not the name of an existing module.

domain_error(current_module)
The argument *Module* is not the name of the current module.

permission_error(directive,module)
The specified directive `end_module/1` is not valid in this position.

Hints

The atom `end_module` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `end_module/0/1` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, begin_module/1, import/1/2, export/1, reexport/1/2, meta/1

Insert a Prolog file in the source text once

:- ensure_loaded +*Filename*
:- ensure_loaded(+*Filename* **)**

The effect of the directive `ensure_loaded/1` can be interpreted as the textual replacement of the directive with the Prolog text in the file *Filename*. The directive will be ignored if the same directive with exactly the same *Filename* as an argument has already been processed. The directive `ensure_loaded/1` can thus be used to load program text only once.

Arguments

Filename	Atom, name of a file

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Filename*.

Hints

The atom `ensure_loaded` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

If *Filename* is not an absolute pathname, the search for the file takes place relative to the directory of the file containing the `ensure_loaded/1` directive.

If *Filename* does not contain the extension *.pro* and if a file of this name cannot be opened for reading, the extension is appended and the resultant filename is used.

Standard

The directive `ensure_loaded/1` complies with the definition of the ISO standard for Prolog.

See also

include/1

Set current error output stream

error_tell(@*Stream*)

The predicate `error_tell/1` makes *Stream* the current error output stream.

If the argument *Stream* is not an input/output stream and not a defined alias, it must be an atom. A file of this name is then opened for writing. If this succeeds, the name is simultaneously registered as an alias for the opened output stream and the stream becomes the current error output stream.

Arguments

Stream	Stream (system-defined term), alias (atom) or filename (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

domain_error(file_name)
The value of the argument *Stream* is not a valid name for a file.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

permission_error(open,source_sink)
The argument *Stream* specifies a file (or a different I/O medium) that cannot be opened.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

Hints

All error messages and warnings of the Prolog system are output on the current error output stream. This stream can always be addressed via the built-in alias `current_error` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

error_telling/1, error_told/0, set_error/1, see/1, tell/1, open/3/4, assign_alias/2

Query current error output stream

error_telling(?*Stream*)

The predicate `error_telling/1` unifies *Stream* with the `current` error output stream.

The current error output stream may be a stream that was set with `error_tell/1` or `set_error/1`.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

domain_error(stream_or_alias)
The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

Hints

The predicate `error_telling/1` can unify *Stream* with an alias or the filename that was possibly used with `error_tell/1` or `open/4`.

All error messages and warnings of the Prolog system are output on the current error output stream. This stream can always be addressed via the built-in alias `current_error` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

error_tell/1, error_told/0, set_error/1, current_error/1, seeing/1, telling/1

Reset current error output stream

error_told

The predicate `error_told/0` resets the default error output stream `user_error` as the current error output stream.

If the earlier current error output stream was not a default stream and is not simultaneously the current output stream, it is closed (see `close/1`).

Hints

All error messages and warnings of the Prolog system are output on the current error output stream. This stream can always be addressed via the built-in alias `current_error` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

error_tell/1, error_telling/1, current_error/2, set_error/1, close/1/2

Raise an exception

exception(@*Error*, @*Info*)

The predicate `exception/2` raises an exception explicitly. Each exception is defined by two criteria:

1. **Type of error** (*Error*)
   Error is a term from the following set:

   `evaluation_error`(*Problem*)
   : An arithmetic expression could not be evaluated without error.

   `domain_error`(*ValidDomain*, *ErrorTerm*)
   : The argument type is correct but the argument does not lie within the predefined value range.

   `existence_error`(*Object*, *ErrorTerm*)
   : An object involved in an operation is not available.

   `instantiation_error`
   : An argument or subterm of an argument is a variable.

   `permission_error`(*Operation*, *Object*, *ErrorTerm*)
   : Execution of an operation on an object is not permitted.

   `representation_error`(*Flag*, *ErrorTerm*)
   : An implementation-related restriction was exceeded.

   `resource_error`(*Resource*)
   : IF/Prolog does not have sufficient resources available to continue with processing.

   `syntax_error`
   : A character string which is to be read as a term does not observe Prolog syntax.

   `system_error`
   : An error involving the operating system occurred.

   `type_error`(*Type*, *ErrorTerm*)
   : An argument or subterm of an argument does not correspond with the expected type.

   Problem, *Range*, *Object*, *Operation*, *Flag*, *Resource* and *Type* are atoms that define the error type in more detail.
   ErrorTerm is a term indicating the argument that triggered the error.

2. **Additional information** (*Info*)
   Info is a list of terms of the type *InfoName* = *Value*, which supply error-specific additional information. *InfoName* is a ground term (usually an atom) and determines the type of additional information; *Value* is any term.

   The IF/Prolog system can supply the following additional information:

`argno`
: *Value* is the argument number of the term which triggered the error, relating to the current Prolog goal (see `goal`).

`culprit`
: *Value* is the term which triggered the error.

`goal`
: *Value* is the goal which triggered the error.

`error`
: *Value* is the atom which determines the type of system error (e.g. could_not_open).

`signal`
: *Value* is the name of the signal which triggered the error.

`interrupted`
: *Value* is the name of a predicate implemented in C that was interrupted by a signal.

`valid_type`
: *Value* is the type of term expected.

`line`
: *Value* is the input line number in case of syntax errors.

`column`
: *Value* is the input column number in case of syntax errors.

`syntax_error`
: *Value* is an atom that indicates the type of syntax error.

`backtrace`
: *Value* is the sequence of goals that led to the error.

The order of the additional information is not defined. *Info* can also be extended to include additional information. The identifiers listed above should only be used according to their meaning.

Arguments

Error	Structure or atom
Info	List containing elements of the type *InfoName* = *Value*
InfoName	Ground term
Value	Term

Exceptions

instantiation_error
: The argument *Error*, *Info* or *InfoName* must not be a variable, but a variable was specified.

type_error(list)
The argument *Info* must be a list, but is a term of another type.

type_error(assignment)
An element of *Info* is not an assignment with the format *InfoName* = *Value*.

Hints

Exceptions are implicitly raised by IF/Prolog when errors occur.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

context/2, exception_handler/3, throw/1

Define local exception handling

exception_handler(+*Goal*, +*Exception*, +*Handler* **)** `[` **@** +*Module* `]`

The predicate `exception_handler/3` executes *Goal* and enables exceptions to be processed.

If *Goal* succeeds, `exception_handler/3` is successfully exited. *Goal* can be re-executed by backtracking.
If *Goal* does not (or does no longer) succeed, `exception_handler/3` fails (i.e. backtracking is initiated).

If an exception occurs while *Goal* is being executed and the exception has the information

`error(` *Error*`,` *Info* `)`

which can be unified with *Exception*, then the subgoal which triggered the exception is **replaced** by *Handler* and execution is continued.
Otherwise, a search is performed for further active contexts which can handle this exception. If no such context can be found, exception handling of IF/Prolog is carried out, i.e. a context jump is executed (see `throw/1`).

Arguments

Goal	Goal
Exception	Variable or structure with the format error(*Term1*,*Term2*)
Handler	Goal

Exceptions

instantiation_error
The argument *Goal* or *Handler* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* or *Handler* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal* or *Handler*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or *Handler* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or *Handler* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

domain_error(error_mask)
The argument *Exception* must be a structure with the format `error(_,_)`, but is a term of another type.

Hints

The predicate `exception_handler/3` is mapped to `context/2`.

The predicate `exception_handler/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

Define handling for all exceptions:

```
[user] ?- exception_handler(goal,
                error(E,I),
                catch_all(error(E,I))).<ENTER>
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

call/1, context/2, exception/2, set_prolog_flag/2

Overlay process

exec(+*Command* **)**

exec(+*Prog*, @*Args* **)**

The predicate `exec/1` starts an operating system dependent command interpreter (UNIX: `sh`) and passes the specified *Command* to this interpreter for execution.

The predicate `exec/2` starts the program *Prog* and passes to it the parameters specified in *Args* as arguments for its execution.

Arguments

Command	Atom
Prog	Atom
Args	List of atoms

Exceptions

instantiation_error
: The argument *Command*, *Prog*, *Args* or an element of *Args* must not be a variable, but a variable was specified.

type_error(atom)
: The argument *Command*, *Prog* or an element of *Args* must be an atom, but is a term of another type.

type_error(list)
: The argument *Args* must be a list, but is a term of another type.

system_error(exec_failed)
: The command *Prog* could not be started. Possibly this is because the command could not be found in the current environment.

Hints

Under UNIX the predicates `exec/1/2` overlay IF/Prolog in the current process with the new program. Other operating systems may create a new process and terminate IF/Prolog.

Under UNIX these predicates can be used in connection with `unix_fork/1` (which duplicates a process) to implement communicating processes.

The predicate `exec/1` is based on the C functions `execl(2)` and `execlp(2)`. The predicate `exec/2` is based on the C function `execvp(2)` (see [10]).

Example

The predicates `exec/1`, `unix_fork/1` and `unix_wait/1` enable a predicate such as `system/1` to be implemented in Prolog:

```
[user] ?- listing.<ENTER>

% *** user: my_system / 1 ***
my_system(Com) :-
    unix_fork(Pid),
    exec_father_son(Pid,Com) .

% *** user: exec_father_son / 2 ***
exec_father_son(0,Com) :-
    !,
    exec(Com) .
exec_father_son(Pid,_) :-
    repeat,
    unix_wait(Pid),
    ! .

yes

[user] ?- my_system('ls -l *.pro').<ENTER>
-rw-rw----   1 brc      prolog        11 Jul 14 09:46 example.pro

yes
```

See also the example in the section *Process management and communication* in the *IF/Prolog User's Guide* [2].

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

unix_fork/1, unix_wait/1/2, unix_kill/2, system/0/1/2/3/5

Transform a grammar rule

expand_term(+*TermIn*, ?*TermOut* **)**

The predicate `expand_term/2` transforms a grammar rule into a normal Prolog term. The predicate is used by the language construct `-->/2`.

If *TermIn* has not the structure of a grammar rule, *TermOut* is unified with the unexpanded *TermIn*.

Arguments

TermIn	Term
TermOut	Term

Example

```
[user] ?- expand_term( (a--> r1; r2), T).

T       = a(_214,_213) :- r1(_214,_213) ; r2(_214,_213) <ENTER>

yes

[user] ?- expand_term( (sentence --> nom_phrase, verb_phrase), T).

T       = sentence(_214,_213) :-
                nom_phrase(_214,_230) , verb_phrase(_230,_213)

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

`-->/2`, phrase/2

Export predicates

:- export @*PredicateList*
:- export(@*PredicateList* **)**

The directive `export/1` makes all the predicates specified in the *PredicateList* available for other modules. They must be made visible there with `import/1/2`.

The directive `export/1` may only occur in the module interface. Predicates declared with the directive `export/1` are also implicitly declared with `private/1`.

Arguments

PredicateList	List with the format `[`*Predicate*,...`]` or structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

permission_error(directive,module)
The specified directive `export/1` is not valid in this position.

Hints

The atom `export` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `export/1` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, import/1/2, meta/1, private/1, reexport/1/2

Initiate backtracking

fail

The language construct `fail/0` always fails, so it can be used deliberately to initiate backtracking.

In the course of backtracking, variable instantiations which have arisen during goal execution are undone, at least up to a point in the search tree at which further solutions can be formed.

Hints

For ANDed (`,/2`) subgoals, subgoals to the right of a `fail/0` subgoal are never executed.

If `fail/0` is called after a predicate which succeeds any number of times in the course of backtracking (e.g. `repeat/0`), an endless loop may result.

The language construct `fail/0` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

Example

The predicate `not/1` can be implemented by the following clauses with `!/0` and `fail/0`:

```
[user] ?- listing.<ENTER>

% *** user: not_new / 1 ***
not_new(Goal) :-
    calling_context(M),
    call(Goal) @ M,
    !,
    fail .
not_new(_).

yes

[user] ?- not_new(true).<ENTER>

no

[user] ?- not_new(fail).<ENTER>

yes
```

Loops can be formed with `fail/0` and predicates which have several solutions.

```
[user] ?- for(1,I,3),<ENTER>
          write(I),nl,<ENTER>
          fail.<ENTER>
1
2
3

no
```

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

!/0, true/0

Check access permission for a file

file_test(+*Pathname*, +*AccessMode* **)**

The predicate `file_test/2` checks whether a file or a directory with the name *Pathname* can be accessed in the way defined by *AccessMode*. If this is possible, `file_test/2` succeeds, otherwise it fails.

There are two possible entries for *AccessMode*:

`read` Test, if *Pathname* can be accessed in read mode, i.e. a file or a directory with this name exists and the user has the necessary permission for read access.

`write` Test, if *Pathname* can be accessed in write mode, i.e. either a file or a directory with this name exists and the user has the necessary permission for write access or the user has the authority in the relevant directory to create files.

Arguments

Pathname	Atom
AccessMode	Atom: read | write

Exceptions

instantiation_error
The argument *Pathname* or *AccessMode* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Pathname* or *AccessMode* must be an atom, but is a term of another type.

domain_error(access_mode)
The argument *AccessMode* does not specify a valid access mode for a file.

Hints

The predicate `file_test/2` does not open any files. It can establish whether a later `open/3/4` could be successful for *Pathname*.

With access mode `read`, the Prolog flag `search_path` will be used as search path for files to be read (see `set_prolog_flag/2`). The same applies for `open/3/4`.

The predicate `file_test/2` is based on the C function `access(2)` (see [10]).

Example

The current directory contains a file called *example.pro* which belongs to the user.

```
[user] ?- file_test('example.pro',read).<ENTER>

yes
```

Users are normally not allowed to create files in the root directory of the file system.

```
[user] ?- file_test('/', write).<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

open/3/4, set_prolog_flag/2

Determine position within an input stream

filepos(@*Stream*, ?*Line* **)**

filepos(@*Stream*, ?*Line*, ?*Column* **)**

The predicate `filepos/2` determines the current line position of the specified input stream *Stream* and unifies the result with *Line*. The current line position is the number of lines processed plus one.

The predicate `filepos/3` additionally determines the character position of the input line and unifies the result with *Column*. The current character position is the number of characters processed in this line plus one.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Line	Integer
Column	Integer

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

type_error(integer)
The argument *Line* or *Column* must be a variable or an integer, but is a term of another type.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

open/3/4

Find all solutions of a goal

findall(?*Term*, +*Goal*, ?*TermList* **)** `[` **@** +*Module* `]`

The predicate `findall/3` unifies *TermList* with a list which contains the same number of instantiations of *Term* as there are solutions for *Goal*. If *Goal* fails, *TermList* is unified with the empty list. After goal execution, all the variables which occur in *Term* and *Goal* are uninstantiated.

If common variables occur in *Term* and *Goal*, the elements of *TermList* will contain each instantiation determined by a solution for these variables.

If *Term* contains variables which do not occur in *Goal*, they are stored uninstantiated in *TermList*.

If *Term* and *Goal* do not contain common variables or if *Term* contains no variables, then *TermList* will contain the term *Term* as many times as *Goal* has solutions.

Arguments

Term	Term
Goal	Goal
TermList	List

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

type_error(list)
The argument *TermList* must be a variable or a list, but is a term of another type.

Hints

The predicate `findall/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

The database contains the following clauses:

```
[user] ?- listing.<ENTER>

% *** user: is_father_of / 2 ***
is_father_of('Zeus','Athena').
is_father_of('Zeus','Apollo').
is_father_of('Cronos','Zeus').
is_father_of('Cronos','Poseidon').

yes
```

You can now find all fathers with:

```
[user] ?- findall(V,is_father_of(V,_),L).<ENTER>

V       = _68
L       = ['Zeus','Zeus','Cronos','Cronos'] <ENTER>

yes
```

And you can find the children with:

```
[user] ?- findall(K,is_father_of(_,K),L).<ENTER>

K       = _68
L       = ['Athena','Apollo','Zeus','Poseidon'] <ENTER>

yes
```

In the following examples, the first argument does not occur in *Goal*:

```
[user] ?- findall(X,member(Y,[a,b,c]),L).<ENTER>

X       = _68
Y       = _69
L       = [_152,_145,_138] <ENTER>

yes

[user] ?- findall(f(a),member(X,[a,b,c]),L).<ENTER>

X       = _70
L       = [f(a),f(a),f(a)] <ENTER>

yes
```

The advantage of being able to insert any desired term for *Term* is that you can then refer to more than one variable in *Goal*.

```
[user] ?- findall(t(X,Y),append(X,Y,[a,b]),L).<ENTER>

X       = _68
Y       = _69
L       = [t([],[a,b]),t([a],[b]),t([a,b],[])] <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

call/1, bagof/3, setof/3, @/2, :/2, set_prolog_flag/2

Test for floating-point number

float(@*TestTerm* **)**

The predicate `float/1` succeeds if *TestTerm* is a floating-point number, otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- float(5.0).<ENTER>

yes

[user] ?- float(5*3).<ENTER>

no

[user] ?- X is 5*3.0, float(X).<ENTER>

X       = 15 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom/1, atomic/1, compound/1, cyclic/1, ground/1, integer/1, nonvar/1, number/1, rational/1, var/1

Query and set floating-point format

float_format(?*FormatOld*, @*FormatNew*)

The predicate `float_format/2` unifies *FormatOld* with the output format set for floating-point numbers at the time of the call and changes the format to that specified in *FormatNew*.

FormatNew must be a structure with two integer arguments L and P.

L specifies the minimum length of the output field. If a negative value is specified for L, the number is output left-justified in the field; if a positive value is specified, the number is output right-justified in the field.

P specifies the number of positions after the decimal point. If $P = 0$, then no decimal point is output. The number to be output is rounded as appropriate.

The following structures are permissible as format specifications:

`e(`L`,`P`)` Exponential notation, i.e. the floating-point number is output in the form `[-]`*d.dddeSdd* (S is either `+` or `-`)
The exponent always has at least two digits.

`'E'(`L`,`P`)` Same as `e(`L`,`P`)`, except that the character `'E'` is used to introduce the exponent.

`f(`L`,`P`)` The number is output as follows: `[-]`*dddd.ddd*

`g(`L`,`P`)` The number is output in the form e(L,P) if, in this form, the exponent is less than -4 or greater than the value of P. Otherwise, the number is output in the form f(L,P). Trailing zeros are removed. The decimal point is output only if it is followed by further digits.

`'G'(`L`,`P`)` Same as `g(`L`,`P`)`, except that the character `'E'` is used to introduce the exponent.

The default setting for the output format is `g(0,6)`.

Arguments

FormatOld	Structure: e(L,P) | f(L,P) | g(L,P) | 'E'(L,P) | 'G'(L,P)
FormatNew	Structure: e(L,P) | f(L,P) | g(L,P) | 'E'(L,P) | 'G'(L,P)
L	Integer or integer expression $-1000 < L < 1000$
P	Integer or integer expression $0 \leq P < 100$

Exceptions

instantiation_error
The argument *FormatNew* or a subterm must not be a variable, but a variable was specified.

type_error(compound)
The argument *FormatNew* must be a structure, but is a term of another type.

domain_error(float_format)
The value of the argument *FormatNew* is not a valid format specification.

type_error(integer)
The argument L or P in *FormatNew* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(compound)
The argument *FormatOld* must be a variable or a structure, but is a term of another type.

domain_error(float_format)
The value of the argument *FormatOld* is not a valid format specification.

type_error(integer)
The argument L or P in *FormatOld* must be a variable or an integer, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument L or P in *FormatNew*, all the exceptions for `is/2` can occur.

Hints

The output format set with `float_format/2` for floating-point numbers affects the output predicates for terms `write/1/2` and possibly also `write_term/2/3`, and conversion predicates (e.g. `write_atom/2`).

The output predicates `write_term/2/3` with the option `quoted(true)`, `write_canonical/1/2`, `writeq/1/2` and `writeq_atom/2` ignore this format so that the re-readability of terms is not impaired.

Example

Output in g format:

```
[user] ?- float_format(F,F).<ENTER>

F       = g(0,6) <ENTER>

yes
```

```
[user] ?- write(1.0e5),nl,<ENTER>
          write(1.0e6),nl.<ENTER>
100000
1e+06

yes
```

Output in e format:

```
[user] ?- float_format(_,e(20,5)),<ENTER>
          write(0.544021437),nl.<ENTER>
         5.44021e-01

yes
```

Output in f format:

```
[user] ?- float_format(_,f(10,8)),<ENTER>
          write(5.44021734e-1),nl.<ENTER>
0.54402173

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

number_chars/2, write_term/2/3, write/1/2

Flush the buffer of an output stream

flush_output

flush_output(@*Stream*)

The predicate `flush_output/0` flushes the contents of the output buffer of the current output stream and clears the buffer.

The predicate `flush_output/1` works in the same way for the specified output stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Hints

The way in which an input/output stream is buffered can be defined when it is opened (see `open/4`).

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Generate a sequence of integers

for(+*Start*, +*Counter*, +*End* **)**
**for(** +*Start*, **?***Counter*, +*End* **)**

The predicate `for/3` generates a sequence of integers in ascending or descending order or checks whether an integer lies within a specified range.

If *Counter* is not instantiated,
then it is unified by backtracking first with *Start*, then with all integers between *Start* and *End*, and finally with *End*.

If *Counter* is already instantiated when the first call is made,
then a check is carried out to determine whether *Counter* lies in the range [*Start*, *End*].

If instantiated, all the arguments are evaluated. Arithmetic expressions which have to return an integer result can therefore be specified.

Arguments

Start	Integer or integer expression
Counter	Variable, integer or integer expression
End	Integer or integer expression

Exceptions

instantiation_error
The argument *Start* or *End* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Start* or *End* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(integer)
The argument *Counter* must be a variable, an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

Furthermore, if an arithmetic expression is specified for the argument *Start*, *End* or *Counter*, all the exceptions for `is/2` can occur.

Example

In the simplest case, the predicate `for/3` generates a sequence of integers:

```
[user] ?- for(2,X,9), write(X), tab(1), fail.<ENTER>
2 3 4 5 6 7 8 9
no
```

Integers in descending order can also be generated.

```
[user] ?- for(2,X,-3), write(X), tab(1), fail.<ENTER>
2 1 0 -1 -2 -3
no
```

If *Counter* is instantiated, it is checked to see whether it lies within the specified range.

```
[user] ?- for(-10,-3,1).<ENTER>

yes
```

The predicate `for/3` can also be used to create iterative loops by backtracking. In the following example, goals are read from a file and called.
Each of these goals must of course have a finite number of solutions, otherwise the system will enter an endless loop.

The input file *example.pro* contains the following Prolog terms:

```
Y is 3+4, write(Y), nl.
true.
fail.
```

Calling `for/3` will yield the following:

```
[user] ?- open('example.pro',read,_,[alias(infile)]).<ENTER>

yes

[user] ?- for(2,X,4), read(infile,Goal),<ENTER>
          write(Goal), nl,<ENTER>
          call(Goal), fail.<ENTER>
_186 is 3 + 4 , write(_186) , nl
7
true
fail

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

fail/0, repeat/0, is/2

Generate or analyze a structure

functor(-*Structure*, +*Functor*, +*Arity* **)**
functor(@*Structure*, ?*Functor*, ?*Arity* **)**

The predicate `functor/3` is used to generate a structure or determine the functor and arity of a structure.

There are two possible cases:

1. *Structure* is not instantiated;
   both *Functor* and *Arity* must be instantiated. If *Arity* is 0, then *Functor* must be a constant and *Structure* is simply unified with *Functor*.

   Otherwise, *Functor* must be an atom and a structure is generated with the functor *Functor* and as many different variables as arguments as are specified by *Arity*. *Structure* is unified with this generated structure.

2. *Structure* is instantiated;
   If *Structure* is a constant (atom or number), *Functor* is unified with this constant and *Arity* with 0.

   If *Structure* is a genuine structure, *Functor* is unified with the functor of the structure and *Arity* with its arity.

Arguments

Structure	Structure or constant
Functor	Constant (atom or number)
Arity	Integer or integer expression $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Functor* or *Arity* must not be a variable, but a variable was specified.

type_error(atomic)
The argument *Functor* must be an atomic term, but is a term of another type.

type_error(atom)
The argument *Functor* must be an atom, but is a term of another type.

type_error(integer)
The argument *Arity* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

representation_error(max_arity)
The argument *Arity* must be an integer within the range 0..127.

domain_error(not_less_than_zero)
The argument *Arity* must be a positive integer, but is negative.

Furthermore, if an arithmetic expression is specified for the argument *Arity*, all the exceptions for `is/2` can occur.

Example

Structure is instantiated:

```
[user] ?- functor(f(X,Y),F,S).<ENTER>

X       = _68
Y       = _69
F       = f
S       = 2 <ENTER>

yes

[user] ?- functor(a,F,S).<ENTER>

F       = a
S       = 0 <ENTER>

yes

[user] ?- functor(g(x),g,1).<ENTER>

yes
```

Structure is uninstantiated:

```
[user] ?- functor(S,f,3).<ENTER>

S       = f(_114,_115,_116) <ENTER>

yes

[user] ?- functor(S,3,0).<ENTER>

S       = 3 <ENTER>

yes
```

`functor/3` is especially useful if structures with a great many arguments are required but only a few of them are instantiated:

```
[user] ?- functor(S,f,7),<ENTER>
          arg(3,S,here).<ENTER>

S       = f(_136,_137,here,_139,_140,_141,_142) <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *Arity* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

arg/3, =../2

Activate garbage collection

garbage_collection(+*Area*)

The predicate `garbage_collection/1` activates the garbage collection for the memory area *Area*. IF/Prolog manages the memory areas `database`, `global`, `constraints`, `local`, `trail` and `extension`. In addition, *Area* can be assigned the value `all`. In this case, garbage collection is performed in all memory areas.

Arguments

Area	Atom: all | database | constraints | global | local | trail | extension

Exceptions

instantiation_error
The argument *Area* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Area* must be an atom, but is a term of another type.

domain_error(memory_area)
The argument *Area* must specify a memory area of the Prolog system. The specified atom is not a valid identifier.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

set_memory_management/3, current_memory_management/3

Query array element

get_array(@*Array*, @*Index*, ?*Term*)

The predicate `get_array/3` queries the element of *Array*, whose index is *Index*, and unifies the element with *Term*.

Arguments

Array	Array (system-defined term)
Index	Positive integer
Term	Term

Exceptions

instantiation_error
The argument *Array* or *Index* must not be a variable, but a variable was specified.

type_error(array)
The argument *Array* must be an array, but is a term of another type.

type_error(integer)
The argument *Index* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(array_size)
The argument *Index* must be a positive integer less than or equal to the number of elements in the array, but is not positive or larger.

Furthermore, if an arithmetic expression is specified for the argument *Index*, all the exceptions for `is/2` can occur.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `get_array/3` is new.

See also

arg/3, array/1/2, create_array/2, set_array/3

Input a byte

get_byte(?*ByteCode*)

get_byte(@*Stream*, **?***ByteCode*)

The predicate `get_byte/1` reads the next byte from the *current* input stream (`current_input`) and unifies *ByteCode* with its integer value.

The predicate `get_byte/2` acts in the same way for the specified input stream *Stream*.

If the end of the input stream has been reached, *ByteCode* is unified with `-1`.

The predicates can be applied only for binary streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
ByteCode	Integer, $-1 \leq ByteCode \leq 255$

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,text_stream)
The current input stream or the argument *Stream* must be a binary stream, but is associated with a text stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

type_error(integer)
The argument *ByteCode* must be a variable or an integer, but is a term of another type.

representation_error(in_byte)
The argument *ByteCode* must be a variable or an integer in the range -1 $\leq$ *ByteCode* $\leq$ 255.

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(input,text_stream)` is generated instead of `permission_error(input,binary_stream)`, when the stream is not a binary stream.

See also

peek_byte/1/2, get_char/1/2, get_code/1/2

Input a character

get_char(?*Character*)

get_char(@*Stream*, ?*Character*)

The predicate `get_char/1` reads the next character from the *current* input stream (`current_input`) and unifies *Character* with the character read in.

The predicate `get_char/2` acts in the same way for the specified input stream *Stream*.

If the end of the input stream has been reached, *Character* is unified with the atom `end_of_file`.

The predicates can be applied only for text streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
Character	Character or `end_of_file`

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,binary_stream)
The current input stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)

system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

type_error(atom)
The argument *Character* must be a variable or an atom, but is a term of another type.

representation_error(in_character)
The argument *Character* must be a variable, a character or the special atom `end_of_file`, but is not representable as a single character.

representation_error(character)
The entity read from the stream is not representable as a single character.

Example

The following predicate is defined for entering words:

```
[user] ?- listing.<ENTER>

% *** user: get_atom / 1 ***
get_atom(A) :-
    get_char(C),
    get_list(C,L),
    atom_chars(A,L),
    ! .

% *** user: get_list / 2 ***
get_list(' ',[]).
get_list('\n',[]).
get_list(end_of_file,[]).
get_list(C,[C|L]) :-
    get_char(C1),
    get_list(C1,L) .

yes
```

Applying this repeatedly to an input line would yield:

```
[user] ?- repeat, get_atom(A).<ENTER>
Input of some Words.<ENTER>

A       = 'Input' ;<ENTER>

A       = of ;<ENTER>

A       = some ;<ENTER>
```

```
A       = 'Words.' <ENTER>

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(input,binary_stream)` is generated instead of `permission_error(input,text_stream)`, when the stream is not a text stream.

See also

peek_char/1/2, get_byte/1/2, get_code/1/2

Input a character

get_code(?*CharCode*)

get_code(@*Stream*, ?*CharCode*)

The predicate `get_code/1` reads the next character from the *current* input stream (`current_input`) and unifies *CharCode* with its integer value.

The predicate `get_code/2` acts in the same way for the specified input stream *Stream*.

If the end of the input stream has been reached, *CharCode* is unified with `-1`.

The predicates can be applied only for text streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
CharCode	Integer

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,binary_stream)
The current input stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

type_error(integer)
The argument *CharCode* must be a variable or an integer, but is a term of another type.

representation_error(in_character_code)
The argument *CharCode* must be a variable, an integer representing a character code or the integer -1.

representation_error(character)
The entity read from the stream is not representable as a single character.

Example

Reading a sequence of bytes up to the end of the line (ASCII code 10):

```
[user] ?- repeat, get_code(B), write(B), nl, B=10.<ENTER>
abcde<ENTER>
97
98
99
100
101
10

B       = 10 <ENTER>

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(input,binary_stream)` is generated instead of `permission_error(input,text_stream)`, when the stream is not a text stream.

See also

peek_code/1/2, get_byte/1/2, get_char/1/2

Get operating system information on files

get_file_info(+*Pathname*, ?*Info* **)**

get_file_info(+*Pathname*, ?*Attribute*, ?*Value* **)**

The predicates `get_file_info/2/3` retrieve information on files from the operating system. Thus the kind and amount of information depends on the current operating system. `get_file_info/2` yields all the information available in a list of the form [*Attribute* = *Value*, ...] whereas `get_file_info/3` allows you to access information more specifically. The predicate fails when you ask for an attribute that is not available for the current operating system or file system.

Attribute	Description
`dev`	ID of device containing file
`ino`	file serial number
`mode`	mode of file (see below)
`nlink`	number of links to file
`uid`	user ID of file
`uname`	user name for user ID or `none`, if not available
`gid`	group ID of file
`gname`	group name for group ID or `none`, if not available
`rdev`	device ID (for character or block special files)
`size`	file size (in bytes) for regular files
`atime`	time of last access
`mtime`	time of last modification
`ctime`	time of last status change
`blksize`	filesystem-specific preferred I/O block size for this object
`blocks`	number of blocks for this object

The Value for `mode` is a list of the following values:

File type	Description
`ifblk`	block special
`ifchr`	character special
`ififo`	FIFO special
`ifreg`	regular
`ifdir`	directory
`iflnk`	symbolic link

File mode	Description
`iread`	file is readable
`iwrite`	file is writable
`iexec`	file is executable
`irusr`	file has read permission for owner
`iwusr`	file has write permission for owner
`ixusr`	file has execute/search permission for owner
`irgrp`	file has read permission for group
`irgrp`	file has write permission for group
`ixgrp`	file has execute/search permission for group
`iroth`	file has read permission for others
`iwoth`	file has write permission for others
`ixoth`	file has execute/search permission for others
`isuid`	set user ID on execution
`isgid`	set group ID on execution
`isvtx`	restricted deletion flag for directories

Arguments

Pathname	Atom, name of a file, directory, etc.
Info	List of the form [*Attribute* = *Value*, ...]
Attribute	Atom, attribute of file
Value	Term, value of attribute

Exceptions

instantiation_error
The argument *Pathname* or a subterm must not be a variable, but a variable was specified.

type_error(atom)
The argument *Pathname* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Pathname* is not a valid name for a file.

type_error(list)
The argument *Info* must be a variable or a list, but is a term of another type.

type_error(atom)
The argument *Attribute* must be a variable or an atom, but is a term of another type.

Hints

To write code that should run under diverse operating systems you can use the predicate `system_name/1` to determine the current operating system.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicates `get_file_info/2/3` are new.

See also

file_test/2, system_name/1

Query the value of a global variable

get_global(+*Name*, ?*Value* **)** `[` **@** +*Module* `]`

The predicate `get_global/2` unifies *Value* with the topmost element of the value stack of the global variable *Name* in the calling module or in the specified *Module*. The value remains the topmost element in the value stack of global variable *Name*.

If the global variable *Name* is defined with `set_global/2`, then its value stack comprises just this one value.

Arguments

Name	Atom, name of a global variable
Value	Term

Exceptions

instantiation_error
The argument *Name* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

existence_error(global_variable)
The argument *Name* is not the name of an existing global variable.

Hints

Global variables are always assigned to a module; they can be neither exported nor imported. In operations with global variables, therefore, only those global variables defined in the calling module or in the specified *Module* are taken into account.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_global/1, pop_global/2, push_global/2, set_global/2, unset_global/1

Determine the cursor position

get_last(?*Line*, ?*Column*)

get_last(@*Stream*, ?*Line*, ?*Column*)

The predicate `get_last/2` supplies the coordinates of the cursor on the screen linked to the current output stream if the device driver for this stream is capable of supplying this information.

If there is no screen linked to the current output stream or if the device driver is not capable of supplying this information, the position set when `move_cursor/2/3` was last called is supplied.

The predicate `get_last/3` acts in the same way as `get_last/2` but uses the screen linked to the specified output stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Line	Integer
Column	Integer

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

type_error(integer)
The argument *Line* or *Column* must be a variable or an integer, but is a term of another type.

Hints

The terminal control predicates are implemented via the appropriate stream control functions of the device drivers (see `stream_control/2`).

The standard device drivers implement screen control via the **terminfo/termcap** service of the operating system. They support only one terminal type for all output streams.

The standard device drivers (see `stream_control/2`) do not support true position queries; instead, they always supply the position set by `move_cursor/2/3`.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

move_cursor/2/3

Get socket option

get_socket_option(@*Socket*, ?*Option*, ?*Value*)

The predicate `get_socket_option/3` gets the current state of an option of a socket.

The argument *Socket* must be a term representing a socket. The arguments ?*Option* and ?*Value* are unified by backtracking with the supported option names and their values.

The set of supported options and corresponding values is system-dependent. The following table contains the allowed values for *Option*. The predicate **get_socket_option/3** fails, if the given option *Option* is not supported for the underlying operating system.

Option	Explanation	Type of *Value*
broadcast	Transmit broadcast messages	on | off
close_on_exec	Control take-over of current sockets	on | off
debug	Record debugging information	on | off
dontroute	Routing bypass for outgoing messages	on | off
error	Get and clear error	atom
keepalive	Keep connections alive	on | off
nonblocking	Make socket asynchroneous/synchroneous	on | off
oobinline	Reception of out-of-band data	on | off
linger	Linger on close if data present	integer
rcvbuf	Size of input buffer	integer
reuseaddr	Local address reuse	on | off
reuseport	Local port reuse	on | off
selectread	Designate socket for reading	on | off
selectwrite	Designate socket for writing	on | off
sndbuf	Size of output buffer	integer
type	Type of the socket	atom

Arguments

Socket	Socket (system-defined term)
Option	Atom
Value	Integer or atom: on | off

Exceptions

instantiation_error
The argument *Socket* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error(atom)
The argument *Option* must be a variable or an atom, but is a term of another type.

domain_error(socket_option)
The argument *Option* is not in the prescribed range of values.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

socket/3, current_socket/3, set_socket_option/3, net_service/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Read up to a specific character

get_until(+*SearchChar*, ?*Text*, ?*EndChar* **)**

get_until(@*Stream*, +*SearchChar*, ?*Text*, ?*EndChar* **)**

The predicate `get_until/3` reads characters from the *current* input stream (`current_input`) until the character *SearchChar* is found or the end of the stream is reached. All the characters read up to that point are concatenated to form an atom and unified with *Text*. The character *SearchChar* which caused the termination of the read operation is not included in the resulting atom.

The predicate `get_until/4` acts in the same way for the specified input stream *Stream*.

The argument *EndChar* is unified with *SearchChar* or the atom `end_of_file`, depending on what led to termination of the read operation.

`end_of_file` can be used as the *SearchChar* so you can read to the end of the stream; when the end is reached, the predicate succeeds. In this case, *EndChar* is unified with `end_of_file`.

The predicates succeed if unification of the result atom with *Text* succeeds and unification of the reason for termination with *EndChar* succeeds. Otherwise the predicates fail.

Arguments

Stream	Stream (system-defined term) or alias (atom)
SearchChar	Character or `end_of_file`
Text	Atom
EndChar	Character or `end_of_file`

Exceptions

instantiation_error
: The argument *Stream* or *SearchChar* must not be a variable, but a variable was specified.

type_error(character)
: The argument *SearchChar* must be a character or the special atom `end_of_file`, but is a term of another type or an invalid atom.

type_error(atom)
: The argument *Text* must be a variable or an atom, but is a term of another type.

type_error(in_character)
: The argument *EndChar* must be a variable, a character or the special atom `end_of_file`, but is a term of another type or an invalid atom.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

Example

Reading a complete input line:

```
[user] ?- get_until('\n',Line,End).<ENTER>
This is one Line.<ENTER>

Line    = 'This is one Line.'
End     = '\n' <ENTER>

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

get_char/1/2

Access a character in an atom

getchar(+*Atom*, +*Position*, ?*Character* **)**

The predicate `getchar/3` unifies *Character* with the *Position*th character in *Atom*.

If *Position* is greater than the length of *Atom* or less than 1, `getchar/3` fails.

Arguments

Atom	Atom
Position	Integer or integer expression
Character	Character

Exceptions

instantiation_error
The argument *Atom* or *Position* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(integer)
The argument *Position* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(character)
The argument *Character* must be a variable or a character, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Position*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- getchar(atom,2,C).<ENTER>

C       = t <ENTER>

yes

[user] ?- getchar('AbCdEfG', 5, W ).<ENTER>

W       = 'E' <ENTER>

yes
```

```
[user] ?- getchar(word,100,C).<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom_concat/3, atom_part/4, atom_prefix/3, atom_suffix/3, index/3, sub_atom/5

Query current working directory

getcwd(?*Dir*)

The predicate `getcwd/1` unifies *Dir* with the full pathname of the current working directory.

Arguments

Dir	Atom

Exceptions

type_error(atom)
The argument *Dir* must be a variable or an atom, but is a term of another type.

Hints

The predicate `getcwd/1` is based on the C function `getcwd(3C)` (see [10]). The structure of pathnames is operating system dependent.

Example

```
[user] ?- getcwd(Dir).<ENTER>

Dir     = '/usr/lib' <ENTER>

yes

[user] ?- chdir('..'), getcwd(Dir).<ENTER>

Dir     = '/usr' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

chdir/1, windows_chdrive/1, windows_getdrive/1

Query environment variable

getenv(+*Name*, ?*Value* **)**

The predicate `getenv/2` unifies *Value* with the value of the environment variable *Name*. If such a variable does not exist, `getenv/2` fails.

Arguments

Name	Atom
Value	Atom

Exceptions

instantiation_error
The argument *Name* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

type_error(atom)
The argument *Value* must be a variable or an atom, but is a term of another type.

Hints

The Prolog system uses the following environment variables:

Variable	Used for
`TMPDIR`	directory for temporary files
`LANG`	national character set supported by the Prolog-scanner
`TERM`	type of the terminal
`HOME`	name of `HOME`-Directory of the user
`SHELL`	name of shell (see `system/0`)
`EDITOR`	name of standard editor (see `edit/0/1`)
`PROROOT`	path where Prolog is installed
`PRORC`	name of startup file
`PROHELP`	name of files with help-information (see `help/0/1`)
`PROPATH`	search path for files used by the Prolog-system

The predicate `getenv/2` is based on the C function `getenv(3C)` (see [10]).

Example

The environment variable `TERM` indicates the type of data display terminal on which the current IF/Prolog process is running, e.g.:

```
[user] ?- getenv('TERM',S).<ENTER>

S       = xterm <ENTER>

yes

[user] ?- getenv(pipapo,X).<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

system/0, default_editor/2, system_parameters/1

Test for ground instantiation

ground(@*TestTerm* **)**

The predicate `ground/1` succeeds if *TestTerm* is ground-instantiated, i.e. contains no variables, otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- ground(X).<ENTER>

no

[user] ?- ground(a(b(c))).<ENTER>

yes

[user] ?- ground([a,b,X,d]).<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom/1, atomic/1, compound/1, cyclic/1, float/1, ground/1, integer/1, nonvar/1, number/1, rational/1, var/1

Terminate Prolog

halt

halt(*+ExitStatus* **)**

The predicates `halt/0/1` immediately terminate Prolog. Any goal execution currently in progress is not completed.

ExitStatus in `halt/1` can be used to send information from the terminated program to the operating system environment (as with `exit(2)` in C).

Arguments

ExitStatus	Integer or integer expression

Exceptions

instantiation_error
The argument *ExitStatus* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *ExitStatus* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

Furthermore, if an arithmetic expression is specified for the argument *ExitStatus*, all the exceptions for `is/2` can occur.

Standard

These predicates comply with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *ExitStatus* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

Display help information

help

help(+*Pattern* **)**

The predicate `help/1` displays suitable help information for the specified *Pattern*, if such information is available.

The argument *Pattern* can be specified as for `match/2/3`, and in most cases will be the name of a predicate.

The predicate `help/0` provides help on `help/1`.

Arguments

Pattern Atom

Exceptions

instantiation_error
The argument *Pattern* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Pattern* must be an atom, but is a term of another type.

permission_error(open,source_sink)
The file containing the help information cannot be opened.

Hints

Search for help information takes place in the files defined by the environment variable `PROHELP`. If the environment variable `PROHELP` does not exist, the file *prolib/system.hlp* under the IF/Prolog root directory (see `proroot/1`) will be used. Help information on all the built-in predicates is supplied in this file. Its structure is described in the beginning of the file.

The environment variable `PROHELP` may contain several filenames separated by the character `:`. The files will be searched in the given order. This enables the user to add own help files to the system.

Example

```
[user] ?- help.<ENTER>

help( +Pattern )
        The predicate help/1 displays suitable help information for the
        specified 'Pattern', if such information is available.
        The argument 'Pattern' can be specified as for match/2/3,
        and in most cases will be the name of a predicate.

        Search for help information takes place in the files defined
        by the environment variable 'PROHELP'.
        If this variable does not exist, the file 'prolib/system.hlp'
        under the IF/Prolog root directory (see proroot/1) will be used.
        Help information on all the built-in predicates is supplied
        in this file.
        Its structure is described in the beginning of the file.

        The environment variable 'PROHELP' may contain several
        filenames separated by the character ':'.
        The files will be searched in the given order.
        This enables the user to add own help files to the system.
help
help( +Pattern )
        System-predicates
        Display help information

yes

[user] ?- help('*abo*').<ENTER>

abolish( @Predicate ) [ @ +Module ]
        System-predicate
        Remove a predicate from the database

abort
        System-predicate
        Return to the toplevel loop of IF/Prolog

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

manual/0/1

Query names or addresses of host

```
# host_addr( +Host, ?Addr )
# host_addr( -Host, +Addr )
```

The predicate `host_addr/2` establishes the relationship between host name and host address. A distinction is made between two cases:

1. *Host* is instantiated;
   The argument *Addr* must be a variable or an atom. It is unified by backtracking with the Internet address(es) of the host whose hostname is *Host*. Normally, a machine has only one Internet address.

2. *Host* is uninstantiated;
   The argument *Addr* must be an atom. The argument *Host* is unified by backtracking with the standard and alias names of the host whose Internet address is *Addr*.

The names and addresses are obtained either from the name server (`named`), the NIS service or the `hosts` database (on UNIX: */etc/hosts* or */etc/inet/hosts*).

Arguments

Host	Atom
Addr	Atom

Exceptions

type_error(atom)
The argument *Host* or *Addr* must be a variable or an atom, but is a term of another type.

domain_error(hostname)
The argument *Host* does not specify a valid hostname.

domain_error(internet_address)
The argument *Addr* does not specify a valid Internet address.

Hints

The sequence of solutions is undefined, if `host_addr/2` has several solutions for the current call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_host/1, net_service/3, socket/3

IF-THEN-ELSE conjunction of goals

if(+*ConditionGoal*, +*ThenGoal*, +*ElseGoal*) [@ +*Module*]

The predicate `if/3` is used for repeated conditional execution.

The predicate is analoguous to *ConditionGoal* `->` *ThenGoal* `;` *ElseGoal*, but in contrast to `->/2`, execution can backtrack over *ConditionGoal*.

The goal succeeds if *ConditionGoal* succeeds and *ThenGoal* succeeds, or if the first execution of *ConditionGoal* fails and *ElseGoal* succeeds.

In procedural view, this means that *ConditionGoal* is first executed. If this is successful, *ThenGoal* is executed. If *ThenGoal* succeeds, the composite goal succeeds as well, else the execution backtracks into *ConditionGoal*. If *ConditionGoal* now succeeds once more, *ThenGoal* is executed again.

If the first execution of *ConditionGoal* fails, *ElseGoal* is executed and the composite goal succeeds if *ElseGoal* succeeds; otherwise it fails.

The subgoal *ThenGoal* is therefore executed if the first execution of *ConditionGoal* succeeds and *ElseGoal* is only executed if it fails.

By backtracking, *ConditionGoal*, *ThenGoal* and *ElseGoal* can have several solutions.

If *ConditionGoal* is a composite goal and if it contains a **cut** (`!/0`), then this cut will have no effect beyond the *ConditionGoal*.

Arguments

ConditionGoal	Goal
ThenGoal	Goal
ElseGoal	Goal

Exceptions

instantiation_error
: The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* must not be a variable, but a variable was specified.

type_error(callable)
: The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
: In executing *ConditionGoal*, *ThenGoal* or *ElseGoal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *ConditionGoal*, *ThenGoal* or *ElseGoal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The call

```
ConditionGoal -> ThenGoal ; ElseGoal
```

is equivalent to

```
if ( (ConditionGoal, !), ThenGoal, ElseGoal) .
```

The predicate `if/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- if(member(A,[a,b]),<ENTER>
| write(A), write(else)).<ENTER>
a
A       = a ;<ENTER>
b
A       = b ;<ENTER>

no

[user] ?- if(member(A,[a,b]),<ENTER>
| (write(A),fail), write(else)).<ENTER>
ab
no
```

```
[user] ?- if(member(x,[a,b]),<ENTER>
| write(x), write(else)).<ENTER>
else
yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `if/3` is new.

See also

->/2, ;/2, !/0, call/1, @/2, :/2, set_prolog_flag/2

Import predicates

:- import +*Module*
:- import(+*Module* **)**

:- import @*PredicateList* **from** +*Module*
:- import(+*Module*, @*PredicateList* **)**

The directives `import/1/2` are used to import predicates from a module.

The directive `import/1` makes all the predicates declared with `export/1` in the module interface of *Module* visible.

The directive `import/2` makes visible only the predicates specified in *PredicateList* from *Module*.

The directive `import/1/2` may only occur in the declaration part of a module body and must precede all predicate definitions in the body.

If the specified *Module* does not exist yet, IF/Prolog attempts to read in the module interface from the file *Module*`.mod`. The file is searched in current search path (see `set_prolog_flag/2`, flag `search_path`).

If the same predicates are imported from different modules, a warning is output. In this case, an import with `import/2` overlays any earlier import, whereas a reimport with `import/1` is ignored.

Arguments

Module	Atom, name of a module
PredicateList	List with the format `[`*Predicate*,...`]` or structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq$ *Arity* ≤ 127

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
: The argument *Module* or *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

existence_error(module)
The argument *Module* is not the name of an existing module.

type_error(list)
The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

existence_error(procedure)
The argument *Predicate* in *PredicateList* specifies a predicate that was not exported from the specified module.

permission_error(directive,module)
The specified directive `import/1/2` is not valid in this position.

Hints

The atom `import` is defined as a prefix operator and the atom `from` is defined as a infix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `import/1/2` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, begin_module/1, end_module/0/1, export/1, reexport/1/2

Include a Prolog file in the source text

:- include +*Filename*
:- include(+*Filename* **)**

The effect of the directive `include/1` can be interpreted as the textual replacement of the directive with the Prolog text in the file *Filename*.

Arguments

Filename	Atom, name of a file

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Filename*.

Hints

The atom `include` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

If *Filename* is not an absolute pathname, the search for the file takes place relative to the directory of the file containing the `include/1` directive.

If *Filename* does not contain the extension *.pro* and if a file of this name cannot be opened for reading, the extension is appended and the resultant filename is used.

Standard

The directive `include/1` complies with the definition of the ISO standard for Prolog.

See also

ensure_loaded/1

Determine the position of a character string in an atom

index(+*Atom*, +*String*, ?*Position* **)**

The predicate `index/3` searches for the first occurrence of *String* in *Atom*. The position at which `index/3` finds *String* is unified with *Position*.

If *String* does not occur in *Atom*, `index/3` fails.

Arguments

Atom	Atom
String	Atom
Position	Integer $0 <$ *Position* $\leq$ Length of *Atom*

Exceptions

instantiation_error
The argument *Atom* or *String* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* or *String* must be an atom, but is a term of another type.

type_error(integer)
The argument *Position* must be a variable or an integer, but is a term of another type.

Example

The string 'here' is contained in the first argument and starts at position 3. Backtracking does not take place.

```
[user] ?- index(xxherexxherexx,here,X).<ENTER>

X       = 3 ;<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom_concat/3, atom_part/4, atom_prefix/3, atom_suffix/3, getchar/3, sub_atom/5, atom_split/3

Initialization goal

:- init +*Goal*
:- init(+*Goal* **)**

:- initialization +*Goal*
:- initialization(+*Goal* **)**

The directives `init/1` and `initialization/1` tell IF/Prolog to execute *Goal* `after` a Prolog text has been loaded.

The directives `init/1` and `initialization/1` may not occur in the module interface.

If there is more than one `init/1` or `initialization/1` directive in a module, the sequence of their execution is undefined.

The goals specified by the directives `init/1` and `initialization/1` are executed after the entire Prolog text has been loaded, even if it contains several modules. The goals are executed in the context of the module, where the directives occurred.

Arguments

Goal	Goal

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

These directives can directly produce the following error messages:

permission_error(directive,module)
The specified directive `init/1` or `initialization/1` is not valid in this position.

Any exception may occur when *Goal* is executed.

Hints

The atoms `init` and `initialization` are defined as prefix operators.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Example

The file *example.pro* contains the following program:

```
:- init((write(init),nl)).

program :- halt.
```

The `init/1` directive is executed after consultation.

```
$ prolog -c example<ENTER>
IF/PROLOG V5.1
Copyright (C) Siemens Nixdorf Information Systems AG, 1996

init
*** consult 'example.pro': loaded in 0.02 sec.
$
```

Standard

The directive `initialization/1` complies with the definition of the ISO standard for Prolog.

The directive `init/1` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

load/1, consult/1, reconsult/1

Test for integer

integer(@*TestTerm* **)**

The predicate `integer/1` succeeds if *TestTerm* is an integer, otherwise it fails.

Arguments

TestTerm	Term

Hints

If IF/Prolog is configured with the extension *rational numbers*, multiple precision integers may be used. The predicate `integer/1` succeeds for them, too.

Example

```
[user] ?- integer( 9 ).<ENTER>

yes

[user] ?- integer( X ).<ENTER>

no

[user] ?- X is 5 - 2, integer( X ).<ENTER>

X       = 3 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom/1, atomic/1, compound/1, cyclic/1, float/1, ground/1, nonvar/1, number/1, rational/1, var/1

Evaluate arithmetic expressions

?*Result* **is @***Expression*
is(?*Result*, **@***Expression* **)**

The predicate `is/2` evaluates the arithmetic expression *Expression* and unifies the result with *Result*. A call of `is/2` can be compared to an assignment statement in other programming languages.

Functions are an important part of arithmetic expressions. They are described in detail in the section 9.4 on *Arithmetic expressions and functions* in this manual. In particular, this section explains when and how the value of an arithmetic expression is formed.

IF/Prolog offers two types of function:

- Arithmetic functions
  These take the values of the arguments and evaluate the result according to mathematical conventions.

- Logical functions
  These are defined only for integer arguments. These arguments are interpreted as bit sequences. The result is always an integer.

The following tables provide an overview of the arithmetic and logical functions which are available in IF/Prolog and which may occur in arithmetic expressions. `I` stands for an integer, `Q` for a rational number, `R` for a floating-point number, `QI` for an integer or rational number, and `N` for any possible number. For functions with names declared as operators, the type of the operator (associativity) is specified in the *Type* column and the precedence of the operator is specified in the *Prec.* column (see `op/3`).

Arithmetic functions with arity 0

Name	Function	Res. type
`cputime`	Used CPU time in seconds	R
`time`	System time in seconds	I
`random`	Random number	I
`pi`	Constant π	R
`dsize`	Total size of the database (bytes)	I
`dused`	Portion of the database used in %	I
`csize`	Total size of the local (control) stack (bytes)	I
`cused`	Portion of the local (control) stack used in %	I
`esize`	Total size of the extended term stack (bytes)	I
`eused`	Portion of the extended term stack used in %	I
`gsize`	Total size of the global stack (bytes)	I
`gused`	Portion of the global stack used in %	I
`tsize`	Total size of the trail stack (bytes)	I
`tused`	Portion of the trail stack used in %	I
`ssize`	Total size of the stacks (bytes)	I
`sused`	Portion of the stacks used in %	I
`maxint`	Largest representable short (32-/64-bit) integer	I
`minint`	Smallest representable short (32-/64-bit) integer	I

Arithmetic functions with arity 1

Functor	Function	Res. type	Type	Prec.
`+`	Neutral function	N = `+` N	fy	200
`-`	Negation	N = `-` N	fy	200

Arithmetic functions with arity 2

Functor	Function	Res. type	Type	Prec.
`+`	Addition	I = I + I	yfx	500
		Q = Q + QI		
		R otherwise		
`-`	Subtraction	I = I - I	yfx	500
		Q = Q - QI		
		R otherwise		
`/`	Division	R = N / N	yfx	400
`//`	Integer division	I = I // I	yfx	400
`rdiv`	Rational division	Q = QI rdiv QI	yfx	400
`mod`	Modulo function	I = I mod I	yfx	400
`rem`	Division remainder	I = I rem I	yfx	400
`*`	Multiplication	I = I * I	yfx	400
		Q = Q * QI		
		Q = Q * Q		
		R otherwise		
`**`	Exponentiation	R = N ** N	xfx	200
`min`	Minimum	N = min(N, N)		
`max`	Maximum	N = max(N, N)		

Arithmetic functions with arity 1, for type conversion

Name	Function	Res. type
`ceiling(N)`	Rounding floating-point number up to the next higher integer	I
`float(N)`	Conversion of integer into floating-point number	R
`floor(N)`	Rounding floating-point number to the next lower integer	I
`round(N)`	Rounding floating-point number to integer	I
`truncate(N)`	Truncating floating-point number to integer	I

Arithmetic functions with arity 1 based on C functions

The *C-funct.* column indicates the C function on which the Prolog function is based.

Name	Function	Res. type	C-Funct.
`abs(N)`	Absolute value	I = abs(I)	abs()
		Q = abs(Q)	
		R = abs(R)	fabs()
`acos(N)`	Arc cosine	R	acos()
`asin(N)`	Arc sine	R	asin()
`atan(N)`	Arc tangent	R	atan()
`cos(N)`	Cosine	R	cos()
`cosh(N)`	Hyperbolic cosine	R	cosh()
`exp(N)`	Exponential function	R	exp()
`log(N)`	Natural logarithm	R	log()
`sign(N)`	Sign function	I = sign(QI)	
		R = sign(R)	
`signum(N)`	Sign function	I = signum(N)	
`sin(N)`	Sine	R	sin()
`sinh(N)`	Hyperbolic sine	R	sinh()
`sqrt(N)`	Square root	R	sqrt()
`tan(N)`	Tangent	R	tan()
`tanh(N)`	Hyperbolic tangent	R	tanh()

Special arithmetic functions for floating-point numbers

Name	Function	Res. type
`float_integer_part(N)`	Integer part of a floating-point number as floating-point number	R
`float_fractional_part(N)`	Fractional part of a floating-point number as floating-point number	R

Logical functions with arity 1

Functor	Function	Res. type	Type	Prec.
\	Complement	I = \ I	fy	200

Logical functions with arity 2

Functor	Function	Res. type	Type	Prec.
`\/`	Bitwise disjunction (OR)	I = I \/ I	yfx	500
`/\`	Bitwise conjunction (AND)	I = I /\ I	yfx	500
`#`	Bitwise exclusive or (XOR)	I = I # I	yfx	500
`<<`	Bitwise left shift	I = I << I	yfx	400
`>>`	Bitwise right shift	I = I >> I	yfx	400

Arguments

Result	Term (usually number)
Expression	Number or arithmetic expression

Exceptions

instantiation_error
The argument *Expression* or a subterm must not be a variable, but a variable was specified.

type_error(evaluable)
The argument *Expression* must be a number or an evaluable arithmetic expression, but is a term of another type, or an operation in the expression *Expression* requires a number as an argument, but a different term was specified.

type_error(integer)
An operation in the expression *Expression* requires an integer as an argument, but a term of another type was specified.

type_error(float)
An operation in the expression *Expression* requires a floating-point number as an argument, but a term of another type was specified.

type_error(number)
An operation in the expression *Expression* requires a number as an argument, but a term of another type was specified.

domain_error(single_precision_integer)
An operation in the expression *Expression* requires an integer with machine precision as an argument, but a big number was specified.

evaluation_error(int_overflow)
An arithmetic overflow was detected during the evaluation of *Expression*.

evaluation_error(float_overflow)
An arithmetic overflow was detected during the evaluation of *Expression*.

evaluation_error(underflow)
An arithmetic underflow was detected during the evaluation of *Expression*.

evaluation_error(undefined)
An arithmetic function with invalid arguments was called during the evaluation of *Expression*.

evaluation_error(zero_divisor)
Division by 0 was performed during the evaluation of *Expression*.

Hints

The atom `is` is defined as an infix operator.

During compilation (`assert...` or `compile/1`) of a predicate containing `is/2`, the arithmetic expressions can be checked and optimized if possible.

Rational numbers, big integers and the operator `rdiv/2` can only be used when configured. Rational numbers can only be used in operations which are suitable for the `Q`-Operand.

Example

Use of operator notation in simple arithmetic expressions:

```
[user] ?- X is 47 + 11.<ENTER>

X       = 58 <ENTER>

yes

[user] ?- A = 6, X is 3 ** 4 - ( 12 / A ).<ENTER>

A       = 6
X       = 79 <ENTER>

yes
```

Use of structure notation in simple arithmetic expressions:

```
[user] ?- is( X, 47 + 11 ).<ENTER>

X       = 58 <ENTER>

yes
```

Use of standard arithmetic functions in arithmetic expressions:

```
[user] ?- X is abs(cos(sqrt(pi))).<ENTER>

X       = 0.200294 <ENTER>

yes
```

Use of zero-arity functions in arithmetic expressions:

```
[user] ?- DF is dsize * (100 - dused) // 100.<ENTER>

DF      = 245141 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The functions `acos/1`, `asin/1`, `cosh/1`, `sinh/1`, `tan/1`, `tanh/1`, `rdiv/2`, `signum/1`, `+/1`, `#/2`, `min/2`, `max/2` and all zero-arity functions are not part of the ISO standard for Prolog. In ISO compatibility mode (see Prolog flag `iso`) none of these functions can be used.

Compatibility

V5.0A The functions `random/0`, `signum/1`, `float_integer_part/1`, `float_fractional_part/1` and `rdiv/2` are new.
The result type of the function `sign/1` has been changed incompatibly.

V5.0B ISO conformance: exception `type_error(evaluable)` is generated instead of `type_error(number)`, when the argument *Expression* is not a number or an evaluable arithmetic expression.

V5.1A The functions `pi/0`, `#/2`, `min/2` and `max/2` are new.

Test for letter

letter(@*TestTerm* **)**

The predicate `letter/1` succeeds if *TestTerm* is an atom which consists of one letter character, otherwise it fails. Which characters are letters, depends on the national language (environment variable `LANG`) **and** on the current language (`current_language/1`).

Arguments

TestTerm	Term

Example

```
[user] ?- letter(a).<ENTER>

yes

[user] ?- letter( '7' ).<ENTER>

no

[user] ?- X = C, letter( X ).<ENTER>

no

[user] ?- X = 'C', letter( X ).<ENTER>

X       = 'C' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

digit/1, current_language/1, set_language/1

Last element in list

list_last(?*Element*, +*List*)

The predicate `list_ast/2` succeeds if *Element* is unifiable with the last element of *List*, otherwise it fails.

Arguments

Element	Term
List	List

Exceptions

instantiation_error
The argument *List* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

domain_error(non_empty_list)
The argument *List* must be a non empty list, but is empty.

Hints

The predicate is semantically equivalent to the following definition:

```
list_last(Element, List) :-
        append(_, [Element], List).
```

but is implemented more efficiently.

Example

Query the last element of list:

```
[user] ?- list_last(X,[c,s,c,a]).<ENTER>

X       = a <ENTER>

yes
```

Instantiate the last element of list:

```
[user] ?- list_last(a,[c,b,c,X]).

X       = a <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `list_last/2` is new.

See also

member/2, memberchk/2, nonmember/2, list_nth/3

Count the elements in a list

list_length(+*List*, ?*Length* **)**

The predicate `list_length/2` unifies *Length* with the number of elements in the list *List*.

Arguments

List	List
Length	Integer

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(integer)
The argument *Length* must be a variable or an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Length* must be a positive integer, but is negative.

Example

```
[user] ?- list_length([nt,tp,brc],L).<ENTER>

L       = 3 <ENTER>

yes

[user] ?- list_length([],L).<ENTER>

L       = 0 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom_length/2

Query and set the list mode of a predicate

list_mode(@*Predicate*, ?*Before*, +*After* **)** `[` @ +*Module* `]`

The predicate `list_mode/3` determines whether or not the predicate *Predicate* may be displayed by `listing/0/1`. The predicates `listing/0/1` only display those predicates for which the list mode is set to `on`. This normally applies only to predicates which have been declared with `dynamic/1` or for which the Prolog flag `debug` was set to `on` when they were defined.

Identification of the predicate is based on its visibility in the calling module or in the specified *Module*.

The argument *Before* is unified with the previously valid mode. The list mode is then set according to *After*.

The list mode can be changed only for predicates for which the modification mode is set to on. Like C predicates, the built-in predicates can never be listed.

If *After* is

`on` then `listing/0/1` may display this predicate

`off` then `listing/0/1` may not display this predicate.

The predicate *Predicate* need not yet be defined. This attribute is retained beyond `abolish/1`.

Arguments

Predicate	Structure with the format *Functor*/*Arity*
Before	Atom: on | off
After	Atom: on | off
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Predicate* or *After* or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
The argument *Predicate* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

type_error(atom)
The argument *After* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *After* must be one of the atoms `on` or `off`, but is another atom.

type_error(atom)
The argument *Before* must be a variable or an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Before* must be a variable or one of the atoms `on` or `off`, but is another atom.

permission_error(access,system_module)
The predicate `list_mode/3` must not be executed for a system module.

Hints

The predicate `list_mode/3` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

A predicate may have certain attributes, which, when it is created, are set implicitly by the system according to the current environment or declarations. They can be modified to a certain extent explicitly by the user. These include the following:

- List mode (see `list_mode/3`)
- Debug mode (see `debug_mode/3`)
- Exception mode (see `predicate_mode/3`)
- Modification mode (see `modify_mode/3`)

These attributes are retained even if the predicate is deleted (with `abolish/1` for example).

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

abolish/1, listing/0/1, dynamic/1, modify_mode/3

Access elements of a list

list_nth(?*Position*, +*List*, ?*Element*)

The predicate `list_nth/3` succeeds if *Element* is unifiable with the element of *List* at the position specified by *Position*. The first list element has the *Position* 1.

The following cases are possible:

1. *Position* is an integer:
   The predicate succeeds if the element of *List* at the position specified by *Position* is unifiable with *Element*. If *Position* is 0 (zero) or greater than the length of *List*, the predicate fails.

2. *Position* is uninstantiated:
   Position is instantiated to the first position, where *Element* is unifiable with an element of *List*. The predicate is not backtrackable, even if *Element* occurs multiply in *List*.

Arguments

Position	Integer or integer expression
List	List
Element	Term

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Position* must be an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Position* must be a positive integer, but is negative.

type_error(list)
The argument *List* must be a list, but is a term of another type.

domain_error(non_empty_list)
The argument *List* must be a non empty list, but is empty.

Hints

The first case of the predicate is semantically equivalent to the following definition:

```
list_nth(1, [Element|_], Element) :- !.
list_nth(Position, [_|Rest], Element) :-
        Position1 is Position - 1,
        list_nth(Position1, Rest, Element).
```

but is implemented more efficiently.

Example

Query the third element of list:

```
[user] ?- list_nth(3,[c,s,c,a],X).<ENTER>

X       = c <ENTER>

yes
```

Instantiate the third element of list:

```
[user] ?- list_nth(3,[c,b,X,c],a).<ENTER>

X       = a <ENTER>

yes
```

Query position of an element:

```
[user] ?- list_nth(N,[b,c,a,c],c).<ENTER>

N       = 2 ; <ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `list_nth/3` is new.

See also

arg/3, member/2, memberchk/2, nonmember/2, list_last/2

List the clauses in the database

listing [@ +*Module*]

listing(+*Functor* **)** [@ +*Module*]
listing(@*Predicate* **)** [@ +*Module*]

The predicate `listing/0` outputs the clauses of all the predicates defined in the calling module or in the specified *Module*, for which the list mode is not `off` (see `list_mode/3`) as formatted clauses to the current output stream.

The variables, if present in the database, are also output with their original names (see `assertz_with_names/3`). If the database does not contain any information on the original names of variables in the listed predicates, then symbolic names are assigned for these variables.

Depending on the argument, the predicate `listing/1` restricts output to the predicates with the functor *Functor* or to the predicate specified by *Predicate*.

If the argument is an atom, then it may also be a pattern as in `match/2/3` and all the predicates matching the pattern are listed.

Arguments

Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Predicate*, *Functor* or *Arity* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Functor* must be an atom, but is a term of another type.

type_error(integer)
The argument *Arity* must be an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Arity* must be a positive integer, but is negative.

representation_error(max_arity)
The argument *Arity* must be an integer within the range 0..127.

Hints

The predicate `listing/0/1` is a metapredicate and considers only predicate definitions which are visible in the calling module or in the specified *Module*.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

list_mode/3, match/2/3, asserta_with_names/2/3, assertz_with_names/2/3, clause_with_names/3

Load compiled Prolog text

load(+*Filename* **)**

The predicate `load/1` loads the Prolog text stored in internal format from the file *Filename*.

If *Filename* does not contain the extension *.cmp* and if a file of this name cannot be opened for reading, the extension is appended and the resultant filename is used.

Such a file may have been created with `compile/1`, `save_module/1` or the `procmp` command and contains either a module interface or a complete module with interface and body.

If there is already a module with the same name as that in *Filename* in the IF/Prolog system, it is deleted before *Filename* is loaded.

Once the module has been loaded, the `init/1` or `initialization/1` directives specified in the body of the module are executed, i.e. the goals specified there are executed in the context of the loaded module.

Arguments

Filename	Atom, name of a file

Exceptions

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Filename*.

domain_error(save_state)
The argument *Filename* specifies a file that does not contain a system state generated by `compile/1`, `save_module/1` or `procmp`.

permission_error(modify,static_procedure)
When loading the file *Filename*, a predicate is to be defined which already exists and which may not be modified.

When initialization directives are executed (`init/1` or `initialization/1`) any exception may arise.

Hints

If a module body of a module contains several `init/1` or `initialization/1` directives, the order in which they are evaluated is undefined upon loading.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

unload/1, compile/1, save_module/1, load_system/1

Restore the old system status

load_system(+*Filename* **)**

The predicate `load_system/1` loads a system state saved with `save_system/1` from *Filename*.

If *Filename* does not contain the extension *.pst* and if a file of this name cannot be opened for reading, the extension is appended and the resultant filename is used.

The current database is completely overwritten in the process.

After `load_system/1`, the goal `user:program` is activated if the predicate `program/0` is defined in the module `user`. Otherwise the input loop of IF/Prolog (`break/0`) is started.

Arguments

Filename	Atom, name of a file

Exceptions

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Filename*.

domain_error(save_state)
The argument *Filename* specifies a file that does not contain a system state generated with `save_system/1` or `save_runtime/1`.

Example

A system state with a user-defined database has been saved with `save_system/1`:

```
[user] ?- listing.<ENTER>

% *** user: tst / 1 ***
tst(1).
tst(2).
tst(3).

yes

[user] ?- save_system(example).<ENTER>

yes
```

This system state, and hence this database also, can be restored in a later IF/Prolog session:

```
[user] ?- load_system(example).<ENTER>

[user] ?- listing.<ENTER>

% *** user: tst / 1 ***
tst(1).
tst(2).
tst(3).

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

save_system/1, load/1

Determine the date and time

localtime(+*Time*, ?*Year*, ?*Month*, ?*Day*, ?*DoW*, ?*DoY*, ?*Hour*, ?*Min*, ?*Sec* **)**

The predicate `localtime/9` converts a system time (see function `time` in `is/2`) into date and time information according to the local time zone.

The individual arguments are unified with the following information (as integers):

Year	Year number	4 digits
Month	Month number	1..12
Day	Day of month	1..31
DoW	Day of week	1..7 (Mon-Sun)
DoY	Day in year	1..366
Hour	Hours	0..23
Min	Minutes	0..59
Sec	Seconds	0..59

Arguments

Time	Integer or integer expression
Year	Integer, $1900 \leq Year$
Month	Integer, $1 \leq Month \leq 12$
Day	Integer, $1 \leq Day \leq 31$
DoW	Integer, $1 \leq DoW \leq 7$
DoY	Integer, $1 \leq DoY \leq 366$
Hour	Integer, $0 \leq Hour \leq 23$
Min	Integer, $0 \leq Min \leq 59$
Sec	Integer, $0 \leq Sec \leq 59$

Exceptions

instantiation_error
: The argument *Time* must not be a variable, but a variable was specified.

type_error(integer)
: The argument *Time* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(integer)
: The argument *Year*, *Month*, *Day*, *DoW*, *DoY*, *Hour*, *Min* or *Sec* must be a variable or an integer, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Time*, all the exceptions for `is/2` can occur.

Hints

The current system time is made available by the zero-arity arithmetic function `time` (mostly as the time in seconds since 1.1.1970 GMT).

If the operating system environment is correctly configured, then conversion by `localtime/9` will take account not only of time zone differences but also of daylight saving time.

The predicate is based on the C function `localtime(3C)` (see [10]).

Example

```
[user] ?- localtime(time,Year,Month,Day,<ENTER>
                    DoW,DinY,Hrs,Min,Sec).<ENTER>

Year    = 1997
Month   = 1
Day     = 8
DoW     = 3
DinY    = 8
Hrs     = 16
Min     = 3
Sec     = 16 <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The argument *Year* is represented with 4 digits

See also

is/2

Convert between lowercase and uppercase letters

lower_upper(+*Lowercase*, ?*Uppercase* **)**
lower_upper(-*Lowercase*, +*Uppercase* **)**

The predicate `lower_upper/2` converts lowercase letters in atoms into uppercase letters, and vice versa.

Arguments

Lowercase	Atom
Uppercase	Atom

Exceptions

instantiation_error
The argument *Lowercase* or *Uppercase* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Lowercase* or *Uppercase* must be an atom, but is a term of another type.

Example

The atom small is converted into 'SMALL':

```
[user] ?- lower_upper(small,X).<ENTER>

X       = 'SMALL' <ENTER>

yes
```

Only the remaining uppercase letters are converted:

```
[user] ?- lower_upper(X,bIG).<ENTER>

X       = big <ENTER>

yes
```

Special characters are not affected:

```
[user] ?- lower_upper(S,'i_am_UPPER_CASE').<ENTER>

S       = i_am_upper_case <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

letter/1

Output manual pages

manual

manual(+*Functor* **)**
manual(@*Predicate* **)**

The predicate `manual/1` writes (on current output) the manual page(s) for the predicates with the functor *Functor* or the specified *Predicate* if such information is available. The predicate succeeds, even if there are no manual pages for the specified predicate.

The manual pages are output for all visible predicates. It is therefore sometimes necessary to `import` the system module containing the predicate.

The predicate `manual/0` outputs this manual page.

Arguments

Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Predicate*, *Functor* or *Arity* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Functor* must be an atom, but is a term of another type.

type_error(integer)
The argument *Arity* must be an integer, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Arity* must be a positive integer, but is negative.

representation_error(max_arity)
The argument *Arity* must be an integer within the range 0..127.

Hints

Files containing the manual pages for all installed components of the IF/Prolog system are located in the IF/Prolog root directory (see `proroot/1`).

Example

```
[user] ?- manual(true).<ENTER>
Name

true/0 - Goal which is always true

Syntax

true

Description

The language construct true/0 succeeds exactly once.

Hints

The language construct true/0 is not a predicate, so it cannot be redefined.
The predicates current_predicate/1 and current_visible/2 do not indicate
language constructs. Explicit module qualification with @/2 or :/2 is not
permitted for language constructs (with the exception of call/1).

Standard

This language construct complies with the definition of the ISO standard for
Prolog.

See also

fail/0

yes

[user] ?- manual(garbona).<ENTER>

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

help/0/1, current_visible/2, import/1/2

Pattern matching

match(+*Mask*, +*Atom* **)**

\# **match(** +*Mask*, +*Atom*, ?*Replacements* **)**

The predicates `match/2/3` carry out pattern matching.

The following special pattern characters can be used to form the mask:

`?` At the place where this character stands in *Mask*, any single character may occur in *Atom*.

`*` At the place where this character stands in *Mask*, any character string (or an empty string) may occur in *Atom*.

other All other characters have their normal significance. Exactly the same character must occur in the corresponding position in *Atom*.

The predicate `match/2` succeeds, if *Atom* can be generated from *Mask* following the resolution of any pattern characters, otherwise it fails.

The predicate `match/3` acts in the same way as `match/2` and, if successful, unifies *Replacements* with a list of all the characters or character strings generated from the pattern characters. Since this assignment does not have to be unique, a search can be carried out for further replacements by backtracking.

Arguments

Mask	Atom
Atom	Atom
Replacements	List of atoms

Exceptions

instantiation_error
The argument *Mask* or *Atom* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Mask* or *Atom* must be an atom, but is a term of another type.

type_error(list)
The argument *Replacements* must be a variable or a list, but is a term of another type.

type_error(atom)
An element of *Replacements* must be a variable or an atom, but is a term of another type.

Hints

As *Mask* usually contains special characters it has to be enclosed in single quotes so that IF/Prolog will recognize the string as an atom.

Example

A question mark within *Mask* is replaced by precisely one character:

```
[user] ?- match('lon?_a?om',long_atom).<ENTER>

yes
```

The pattern character '*' stands for any string of characters. The list of individual replacements is unified with *L*:

```
[user] ?- match('long?a*',long_atom,L).<ENTER>

L       = ['_',tom] <ENTER>

yes

[user] ?- match('*a*',abrakadabra,L).<ENTER>

L       = ['',brakadabra] ;<ENTER>

L       = [abr,kadabra] ;<ENTER>

L       = [abrak,dabra] ;<ENTER>

L       = [abrakad,bra] ;<ENTER>

L       = [abrakadabr,''] ;<ENTER>

no
```

The predicate `suffix_pro(`*Name, New_Name*`)` checks whether *Name* contains *.pro* as its extension. If not, this extension is added. The result is unified with *New_Name*.

```
[user] ?- listing.<ENTER>

% *** user: suffix_pro / 2 ***
suffix_pro(Name,Name) :-
    match('*.pro',Name),
    ! .
suffix_pro(Name,New_Name) :-
    atom_concat(Name,'.pro',New_Name) .
```

```
yes

[user] ?- suffix_pro(file,F).<ENTER>

F       = 'file.pro' <ENTER>

yes

[user] ?- suffix_pro('file.pro',F).<ENTER>

F       = 'file.pro' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

match_atom/2, regexp/2/3

Concatenate matching atoms to form an atom

match_atom(@*List*, ?*Atom* **)**
match_atom(+*List*, +*Atom* **)**

The predicate `match_atom/2` succeeds, if the concatenation of the elements in *List* can be unified with *Atom*.

If all elements of *List* are atoms, the predicate acts in the way as `concat_atom/2`. If any elements of *List* are variables, IF/Prolog finds by backtracking such atoms that the concatenation of the elements forms the atom *Atom*.

The predicate `match_atom/2` can be used for simple pattern matching instead of `match/2`.

Arguments

List	List of atoms or variables
Atom	Atom

Exceptions

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(atom)
An element of *List* must be an atom, but is a term of another type.

instantiation_error
The argument *Atom* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

Example

```
[user] ?- match_atom([A,B,C],abc).<ENTER>

A       = ''
B       = ''
C       = abc ;<ENTER>

A       = ''
B       = a
C       = bc ;<ENTER>

A       = a
```

```
B       = ''
C       = bc ;<ENTER>

A       = ''
B       = ab
C       = c ;<ENTER>

A       = a
B       = b
C       = c ;<ENTER>

A       = ab
B       = ''
C       = c ;<ENTER>

A       = ''
B       = abc
C       = '' ;<ENTER>

A       = a
B       = bc
C       = '' ;<ENTER>

A       = ab
B       = c
C       = '' ;<ENTER>

A       = abc
B       = ''
C       = '' ;<ENTER>

no

[user] ?- match_atom([A,'+',B,'=',C], '2+3=5').

A       = '2'
B       = '3'
C       = '5' <ENTER>

yes

[user] ?- match_atom([p,a,n,g],pang).<ENTER>

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

match/2/3, regexp/2/3, concat_atom/2/3

List membership

member(?*Element*, ?*List*)

The predicate `member/2` establishes the relationship between lists and their elements. The following cases are possible:

1. *Element* and *List* are instantiated;
   the predicate `member/2` checks whether *Element* is contained in *List*. If so, `member/2` succeeds during backtracking as often as *Element* occurs in *List*. Otherwise `member/2` fails.

2. *Element* is uninstantiated and *List* is instantiated;
   `member/2` fails if *List* is the empty list. Otherwise, when the goal is first executed the first element from *List* is instantiated to *Element*.

   During each backtracking cycle, *Element* is instantiated to the next element of *List*.

3. *Element* is instantiated and *List* is uninstantiated;
   the predicate `member/2` unifies *List* in the first call with a list containing *Element* as its first element and a tail variable. In each backtracking cycle, *Element* appears as the second, third, etc. element of *List*.

4. *Element* and *List* are uninstantiated;
   the predicate `member/2` generates variable masks which fulfill the member relationship.

Please note that, due to the recursive structure of the predicate, various combinations of the above four cases may occur.

Arguments

Element	Term
List	List

Hints

It is possible for the predicate `member/2` to have an infinite number of solutions, e.g. if it is called with two uninstantiated variables.

Calls such as `member(Y,Y)` produce cyclic list structures which can lead to an endless loop if, for example, they are to be output.

Example

The simplest application involves a check as to whether an element is contained in a list.

```
[user] ?- member(a,[c,s,c,a]).<ENTER>

yes
```

The predicate can also be used to query the elements of a list one at a time.

```
[user] ?- member(X,[a,b,c]).<ENTER>

X       = a ;<ENTER>

X       = b ;<ENTER>

X       = c ;<ENTER>

no
```

In addition, the predicate can generate "variable masks", as in the case of `append/3`.

```
[user] ?- member(a,L).<ENTER>

L       = [a|_113] ;<ENTER>

L       = [_112,a|_115] ;<ENTER>

L       = [_112,_114,a|_117] ;<ENTER>

L       = [_112,_114,_116,a|_119] <ENTER>

yes
```

An empty list contains nothing:

```
[user] ?- member(X,[]).<ENTER>

no
```

Applications where both arguments are only partially instantiated are also possible:

```
[user] ?- member(f(X),[a,f(Y),c|Z]).<ENTER>

X       = _68
Y       = _68
Z       = _74 ;<ENTER>
```

```
X       = _68
Y       = _71
Z       = [f(_68)|_135] ;<ENTER>

X       = _68
Y       = _71
Z       = [_134,f(_68)|_137] ;<ENTER>

X       = _68
Y       = _71
Z       = [_134,_136,f(_68)|_139] <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

memberchk/2, nonmember/2, append/3, connect/3, list_last/2, list_nth/3

List membership

memberchk(?*Element*, ?*List*)

The predicate `memberchk/2` succeeds if *Element* occurs in *List*, otherwise it fails.

The following cases are possible:

1. *List* is uninstantiated:
   the predicate instantiates *List* to a list containing *Element* as its only element.
2. *List* is instantiated:
   the predicate checks whether *Element* is unifiable with an element in *List*.

The predicate should be used only when *Element* and *List* are sufficiently instantiated.

Arguments

Element	Term
List	List

Exceptions

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

Hints

The predicate is semantically equivalent to the following definition:

```
memberchk(Element, List) :-
        member(Element, List), !.
```

but is implemented more efficiently.

Example

The simplest application involves a check as to whether an element is contained in a list.

```
[user] ?- memberchk(a,[c,s,c,a]).<ENTER>

yes
```

The list must not necessarily be ground-instantiated. As soon as a unifiable element is found, the predicate succeeds deterministically:

```
[user] ?- memberchk(a,[c,X,c,a]).<ENTER>

X       = a ;<ENTER>

no
```

An empty list contains nothing:

```
[user] ?- memberchk(X,[]).<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `memberchk/2` is new.

See also

member/2, nonmember/2

Declare metapredicates

:- meta @*PredicateList*
:- meta(**@***PredicateList* **)**

The directive `meta/1` is used to declare the predicates specified in *PredicateList* as metapredicates.

The directive `meta/1` may occur in the module interface and in the declaration part of a module body. If `meta/1` occurs in the module interface, it implies `export/1` for the specified predicates.

If the directive `meta/1` occurs in the module body, it must appear before the first predicate definition.

Predicates declared with the directive `meta/1` are also implicitly declared by `private/1`.

Metapredicates are automatically executed in the context of the calling module, instead of the normal practice of executing predicates in the context of the defining module. All the metapredicates can be qualified in the call by an explicit definition of the module context in which they are to be executed (see `@/2`). For this, they are implicitly passed an additional argument which contains the relevant module name.

For most predicates, the context in which they are executed is not important, but `asserta/1`, for example, should normally enter facts and rules into the database in the module from which it was called and not in the module `system` where it is defined. All the built-in predicates which have module-specific side effects (such as `asserta/1/2`), which use module-specific data (such as `write/1/2`) or which themselves call goals (such as `findall/3`) are therefore defined as metapredicates.

Arguments

PredicateList	List with the format `[`*Predicate*,...`]` or structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

permission_error(directive,module)
The specified directive `meta/1` is not valid in this position.

Hints

The atom `meta` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `meta/1` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, begin_module/1, end_module/0/1, import/1/2, export/1, reexport/1/2, private/1, calling_context/1

Query and set the modify mode of a predicate

modify_mode(@*Predicate*, ?*Before*, +*After* **)** `[` @ +*Module* `]`

The predicate `modify_mode/3` defines whether or not the predicate *Predicate* is modifiable, i.e. whether its clauses in the database and its attributes may be manipulated.

Identification of the predicate is based on its visibility in the calling module or in the specified *Module*.

The argument *Before* is unified with the previously valid mode. The modification mode is then set according to *After*.

Like C predicates, built-in predicates cannot be modified.

If *After* is

`on` then the definition of *Predicate* can be dynamically modified and the predicate attributes can be modified.

`off` then the definition of *Predicate* and the predicate attributes cannot be modified.

The predicates for manipulating the database (`asserta/1/2`, `retract/1/2`, etc.) may only manipulate predicates for which the modification mode is set to on (except `abolish/1`). This normally applies only to predicates which have been declared with `dynamic/1` or for which the Prolog flag `debug` was set to `on` when they were defined.

The predicate *Predicate* need not yet be defined. This attribute is retained beyond `abolish/1`.

Arguments

Predicate	Structure with the format *Functor*/*Arity*
Before	Atom: on | off
After	Atom: on | off
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Predicate* or *After* or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
The argument *Predicate* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

type_error(atom)
The argument *After* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *After* must be one of the atoms `on` or `off`, but is another atom.

type_error(atom)
The argument *Before* must be a variable or an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Before* must be a variable or one of the atoms `on` or `off`, but is another atom.

permission_error(access,system_module)
The predicate `modify_mode/3` must not be executed for a system module.

Hints

The predicate `modify_mode/3` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

A predicate may have certain attributes, which, when it is created, are set implicitly by the system according to the current environment or declarations. They can be modified to a certain extent explicitly by the user. These include the following:

- List mode (see `list_mode/3`)
- Debug mode (see `debug_mode/3`)
- Exception mode (see `predicate_mode/3`)
- Modification mode (see `modify_mode/3`)

These attributes are retained even if the predicate is deleted (with `abolish/1` for example).

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

abolish/1, dynamic/1, debug_mode/3, list_mode/3, predicate_mode/3

Define a module

:- module +*Module*
:- module(+*Module* **)**

The directive `module/1` defines a new module and marks the start of the module interface.

The module interface ends with an `end_module/0/1` directive, the start of the module body (`begin_module/1`), the start of a new module interface (`module/1`) or the end of the file.

If there is already a module with the name *Module* which is not a system module, this module is first deleted. The interface cannot be redefined for the built-in module `user`.

The interface of a module may only contain the directives permitted there, namely `char_conversion/2`, `export/1`, `reexport/1/2`, `meta/1`, `op/3` or `include/1`.

Arguments

Module	Atom

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *Module* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

permission_error(modify,system_module)
The argument *Module* is the name of a system module. System modules may not be modified.

permission_error(modify,module)
The argument *Module* is the name of a module with an interface which cannot be modified (e.g. `user`).

Hints

The atom `module` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `module/1` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

begin_module/1, end_module/0/1, export/1, meta/1, import/1/2

Position the cursor

move_cursor(+*Line*, +*Column* **)**

move_cursor(@*Stream*, +*Line* +*Column* **)**

The predicate `move_cursor/2` moves the cursor to the specified *Line* and *Column* position of the screen associated with the current output stream.

The predicate `move_cursor/3` acts in the same way as `move_cursor/2`, but uses the screen associated with the specified output stream *Stream*.

If there is no screen associated with the current output stream or if the device driver is able to execute this operation, the predicate has no effect, but the call succeeds nevertheless.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Line	Integer or integer expression
Column	Integer or integer expression

Exceptions

instantiation_error
The argument *Stream*, *Line* or *Column* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Line* or *Column* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

Furthermore, if an arithmetic expression is specified for the argument *Line* or *Column*, all the exceptions for `is/2` can occur.

Hints

The terminal control predicates are implemented via the appropriate stream control functions of the device drivers (see `stream_control/2`).

The standard device drivers implement screen control via the **terminfo/termcap** service of the operating system. They support only one terminal type for all output streams.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

get_last/2/3, tty_size/2/3

Declare predicates as being scattered over several files

:- multifile @*PredicateList*
:- multifile(@*PredicateList* **)**

The clauses of predicates declared with the directive `multifile/1` can be scattered over several files, i.e. when files are loaded using `consult/1`, each predicate definition is extended. To do this, the directive `multifile/1` behaves in exactly the same manner as the directive `dynamic/1`.

The directive `multifile/1` may only occur in the declaration part of a module body. The first `multifile/1` directive indicating a predicate must precede all clauses for this predicate.

Arguments

PredicateList	List with the format `[`*Predicate*,...`]` or structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

permission_error(directive,module)
The specified directive `multifile/1` is not valid in this position.

Hints

The atom `multifile` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `multifile/1` complies with the definition of the ISO standard for Prolog.

See also

consult/1, dynamic/1, discontiguous/1

Query communication service

net_service(+*Service*, ?*Protocol*, ?*Port* **)**

The predicate `net_service/3` queries the protocol name and port number of a TCP/IP service.

The argument *Service* specifies the name of the TCP/IP service.

If *Protocol* is instantiated, it must be either `tcp` or `udp`, representing stream or datagram protocol. Otherwise, the argument *Protocol* is unified with the protocol name of the service. The predicate is not backtrackable, even if there are both `tcp` and `udp` services for the specified *Service*.

The argument *Port* is unified with the port number of the service.

The information is obtained either from the NIS service or the `services` database (on UNIX: */etc/services* or */etc/inet/services*).

Arguments

Service	Atom
Protocol	Atom: tcp | udp
Port	Integer

Exceptions

instantiation_error
The argument *Service* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Service* must be an atom, but is a term of another type.

domain_error(service)
The argument *Service* does not specify an existing TCP/IP service.

type_error(atom)
The argument *Protocol* must be a variable or an atom, but is a term of another type.

domain_error(protocol)
The argument *Protocol* does not specify a valid TCP/IP protocol.

type_error(integer)
The argument *Port* must be a variable or an integer, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_host/1, host_addr/2, socket/3

Output newline

nl

nl(@*Stream* **)**

The predicate `nl/0` outputs the new line character on the *current* output stream (`current_output`).

The predicate `nl/1` has the same effect, but on the specified output stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Hints

Calling `nl/0` is equivalent to `write('\n')` or `put_char('\n')`. The effect of `nl/1` is analogous.

Example

Generating a newline with `nl/0`:

```
[user] ?- write(peter), nl, write(king).<ENTER>
peter
king
yes
```

Generating a newline with `'\n'`:

```
[user] ?- write('Hyphen-\nation').<ENTER>
Hyphen-
ation
yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

See also

put_char/1/2, write/1/2

List membership

nonmember(@*Element*, @*List*)

The predicate `nonmember/2` succeeds if *Element* does **not** occur in *List*, otherwise it fails.

The argument *List* must be a list. The predicate checks whether *Element* is unifiable with an element in *List*. If no unifiable element is found, the predicate succeeds. It would be meaningless to instantiate *Element* in any way, since there are infinitely many terms that do not occur in *List*.

The predicate should be used only when *Element* and *List* are sufficiently instantiated.

Arguments

Element	Term
List	List

Exceptions

type_error(list)
The argument *List* must be a list, but is a term of another type.

Hints

The predicate is semantically equivalent to the following definition:

```
nonmember(Element, List) :-
        not memberchk(Element, List).
```

but is implemented more efficiently.

Example

The simplest application involves a check as to whether an element is not contained in a list.

```
[user] ?- nonmember(x,[c,s,c,a]).<ENTER>

yes
```

The list must not necessarily be ground-instantiated. As soon as a unifiable element is found, the predicate fails:

```
[user] ?- nonmember(a,[c,X,y,z]).

no
```

An empty list contains nothing:

```
[user] ?- nonmember(X,[]).<ENTER>

X       = _154

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `nonmember/2` is new.

See also

member/2, memberchk/2

Do not output load message

```
:- nonotify
```

The directive `nonotify/0` disables the output of a message, when the Prolog text is read in with `consult/1`, `reconsult/1` or `compile/1`. The effect is local, in contrast to the Prolog flag `consult`, i.e. applies only for the Prolog text containing the directive and files included from this text.

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

Hints

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `nonotify/0` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

consult/1, ensure_loaded/1, include/1, compile/1, set_prolog_flag/2

Test for instantiation

nonvar(@*TestTerm* **)**

The predicate `nonvar/1` succeeds if *TestTerm* is not a variable, otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- nonvar( a(X,Y) ).<ENTER>

X       = _68
Y       = _69 <ENTER>

yes

[user] ?- nonvar(three).<ENTER>

yes

[user] ?- nonvar( X ).<ENTER>

no

[user] ?- Y is 35, nonvar( Y ).<ENTER>

Y       = 35 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom/1, atomic/1, compound/1, cyclic/1, float/1, ground/1, integer/1, number/1, rational/1, var/1

Negation through failure

not @*Goal*
not(@*Goal* **)**

The language construct `not/1` calls *Goal* and fails if *Goal* succeeds. Otherwise, `not/1` succeeds.

Arguments

Goal Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The atom `not` is defined as a prefix operator.

The language construct `not/1` is identical to `\+/1`.

The language construct `not/1` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

The predicates activated in the goal must be visible in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed

in the context of the specified *Module*, unless the `@/2` qualification is used for a meta-predicate to indicate explicitly the module context in which this predicate is to be executed.

Example

The language construct `not/1` can be used to call a goal without variables being instantiated:

```
[user] ?- not not member(Y, [a, b]).<ENTER>

Y       = _68 <ENTER>

yes
```

Standard

This language construct is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

\+/1, call/1, fail/0, @/2, :/2, set_prolog_flag/2

Test for number

number(@*TestTerm* **)**

The predicate `number/1` succeeds if *TestTerm* is an integer or a floating-point number, otherwise it fails.

Arguments

TestTerm	Term

Hints

If IF/Prolog is configured with the extension *rational numbers*, rational numbers and multiple precision integers may be used. The predicate `number/1` succeeds for them, too.

Example

```
[user] ?- number(353535).<ENTER>

yes

[user] ?- number(4.2).<ENTER>

yes

[user] ?- number(-1234.0e+222).<ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom/1, atomic/1, compound/1, cyclic/1, float/1, ground/1, integer/1, nonvar/1, rational/1, var/1

Convert between a number and a list of characters

number_chars(+*Number*, ?*List* **)**
number_chars(-*Number*, @*List* **)**

The predicate `number_chars/2` establishes the relationship between a number and the list comprising of the individual characters of the number. A distinction is made between two cases:

1. *Number* is instantiated:
   List must be either a variable or a list of variables or characters. The predicate `number_chars/2` creates the list of characters in *Number* and unifies the result with *List*. If *Number* is an arithmetic expression, the expression is evaluated first, and the result is then converted into characters.

2. *Number* is uninstantiated:
   List must be a completely instantiated list of characters. The predicate `number_chars/2` creates a number from *List* in accordance with the characters in *List*. The result is unified with *Number*.

Arguments

Number	Number or arithmetic expression
List	List of characters

Exceptions

instantiation_error
The argument *List* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(character)
An element of *List* must be a character, but is a term of a another type.

syntax_error('illegal number')
The characters in *List* do not represent a number.

type_error(number)
The argument *Number* must be a number or an arithmetic expression, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

Hints

The representation of floating-point numbers does **not** depend on the floating-point format currently set but on the maximum precision.

Example

```
[user] ?- number_chars(1234,X).<ENTER>

X       = ['1','2','3','4'] <ENTER>

yes
```

The character list can also be in hexadecimal notation:

```
[user] ?- number_chars(X,['0','x','1','F']).<ENTER>

X       = 31 <ENTER>

yes
```

but

```
[user] ?- number_chars(31,['0','x','1','F']).<ENTER>

no
```

Floating-point numbers can also be processed:

```
[user] ?- number_chars(X,['1','.','2']).<ENTER>

X       = 1.2 <ENTER>

yes
```

The conversion does not depend on the floating-point format currently set:

```
[user] ?- float_format(F,F),<ENTER>
          number_chars(1.2,L).<ENTER>

F       = g(0,6)
L       = ['1','.','1','9','9','9','9','9','9','9','9','9','9','9','9',
'9','9','9','9'] <ENTER>

yes

[user] ?- float_format(_,e(0,1)),<ENTER>
          float_format(F,F),<ENTER>
```

```
          number_chars(1.2,L).<ENTER>

F       = e(0,1)
L       = ['1','.','1','9','9','9','9','9','9','9','9','9','9','9',
'9','9','9','9'] <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *Number* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

atom_chars/2, number_codes/2

Convert between a number and a list of character codes

number_codes(+*Number*, ?*List* **)**
number_codes(-*Number*, @*List* **)**

The predicate `number_codes/2` establishes the relationship between a number and the list comprising of the individual character codes of the number. A distinction is made between two cases:

1. *Number* is instantiated:
   List must be either a variable or a list of variables or character codes. The predicate `number_codes/2` creates the list of character codes in *Number* and unifies the result with *List*. If *Number* is an arithmetic expression, the expression is evaluated first, and the result is then converted into character codes.

2. *Number* is uninstantiated:
   List must be a completely instantiated list of character codes. The predicate `number_codes/2` creates a number from *List* whose external representation matches the sequence of character codes in *List*. The result is unified with *Number*.

Arguments

Number	Number or arithmetic expression
List	List of integers or integer expressions

Exceptions

type_error(number)
The argument *Number* must be a number or an arithmetic expression, but is a term of another type.

instantiation_error
The argument *List* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(integer)
An element of *List* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

representation_error(character_code)
An element of *List* must be an integer representing a character code.

syntax_error('illegal number')
The characters in *List* do not represent a number.

Furthermore, if an arithmetic expression is specified for the argument *Number* or one element of *List*, all the exceptions for `is/2` can occur.

Hints

Integer expressions in *List* will only be evaluated if *Number* is not instantiated.

The representation of floating-point numbers does **not** depend on the floating-point format currently set but on the maximum precision.

Example

```
[user] ?- number_codes(1234,X).<ENTER>

X       = [49,50,51,52] <ENTER>

yes
```

Floating-point numbers can also be processed:

```
[user] ?- number_codes(X,[49,46,50]).<ENTER>

X       = 1.2 <ENTER>

yes
```

The conversion does not depend on the floating-point format currently set:

```
[user] ?- float_format(F,F),<ENTER>
          number_codes(1.2,L).<ENTER>

F       = g(0,6)
L       = [49,46,49,57,57,57,57,57,57,57,57,57,57,57,57,
57,57,57] <ENTER>

yes

[user] ?- float_format(_,e(0,1)),<ENTER>
          float_format(F,F),<ENTER>
          number_codes(1.2,L).<ENTER>

F       = e(0,1)
L       = [49,46,49,57,57,57,57,57,57,57,57,57,57,57,57,
57,57,57] <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the arguments *Number* and *List* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for these predicates. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

Compatibility

V5.0B Inconsistency in the ISO standard has been fixed: exception `type_error(list)` or `representation_error(character_code)` is generated instead of `domain_error(character_code_list)`, when the argument *List* is not a list of character codes.

See also

atom_codes/2, number_chars/2, char_code/2

Activate a goal once only

once(+*Goal* **)** `[` **@** +*Module* `]`

The predicate `once/1` behaves like `call/1`, but cannot be backtracked even if the specified *Goal* is re-executable.

Arguments

Goal	Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate `once/1` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- once(member(S,[1,2,3])).<ENTER>

S       = 1 ;<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

call/1, !/0, @/2, :/2, set_prolog_flag/2

Define and delete operators

op(+*Priority*, +*Assoc*, @*Names* **)**

The predicate `op/3` can be used to define operators and to delete operator definitions.

Operators permit terms to be entered or displayed in operator notation instead of the normal structure notation. This enables predicates to be represented in a more readable format.

If *Names* is an atom, the atom is defined or deleted as an operator. If *Names* is a list of atoms, the atoms in the list are defined or deleted as operators in the given order.

The operator is assigned the specified *Priority*, which is represented by a positive integer between 1 and 1200. If a value of 0 is specified as the *Priority*, the operator is deleted if it already exists.

The operator has the specified associativity *Assoc*. If operators of the same precedence occur together, *Assoc* determines how the operands are associated. *Assoc* also determines whether the operator is a prefix, an infix, or a postfix operator. The following atoms are used to specify the type:

`fx`	Prefix	non-associative
`fy`	Prefix	right-associative
`xf`	Postfix	non-associative
`yf`	Postfix	left-associative
`xfx`	Infix	non-associative
`xfy`	Infix	right-associative
`yfx`	Infix	left-associative

Arguments

Priority	Integer or integer expression $0 \leq Priority \leq 1200$
Assoc	Atom: xf | yf | fx | fy | xfx | xfy | yfx
Names	Atom or list of atoms

Exceptions

instantiation_error
: The argument *Priority*, *Assoc* or *Names* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Priority* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(operator_priority)
The argument *Priority* is not in the prescribed range of values for the precedence of operators.

type_error(atom)
The argument *Assoc* or an element of *Names* must be an atom, but is a term of another type.

domain_error(operator_specifier)
The argument *Assoc* is not a valid type for an operator definition.

type_error(list)
The argument *Names* must be a regular list or an atom, but is a term or another type or not regular.

permission_error(modify,operator)
The argument *Names* specifies an existing operator, whose definition is incompatible with the new definition.

permission_error(create,operator)
The argument *Names* specifies an existing operator that may not be redefined.

Hints

The section 9.3.1 on *Converting expressions into equivalent normal structures* contains a detailed description of the significance of the precedence and the type of an operator when an expression is converted into an equivalent normal structure.
A list of built-in operators can be found in this manual in the section 9.3.2.

Different operators may have the same name; they are then distinguished on the basis of their type.
A prefix operator and an infix operator with the same name are different and may have different priorities. For example, IF/Prolog has two different built-in operators with the name `'-'`: the subtraction operator (defined as `op(500,yfx,+)`) and the negation operator (defined as `op(200,fy,+)`).

If expressions with operators of the same precedence are used in the argument area of a non-associative operator when entering terms in operator notation, and if there are no parentheses to provide an unambiguous operator/argument relationship, a syntax error is reported (e.g. `a < b = a < b`).

A syntax error is also reported if there is a conflict in the associativity information (e.g. if a left-associative operator precedes a right-associative operator with the same precedence). Such problems can be avoided by using the appropriate parentheses.

Operators are not associated with a module, but are globally visible.

Example

An exclusive OR operator is to be defined. The atom `'#'` is to be reserved for this operation. A typical file named *example.pro* could then look like this:

```
:- op(1100,xfy,#).

Goal1 # Goal2:- not Goal1, !, Goal2.
Goal1 # Goal2:- Goal1, not Goal2.
```

The `op/3` directive ensures that the operator can be used in the input file. Since the operator remains available after the load operation, it can be used after loading with `consult/1`:

```
[user] ?- [example].<ENTER>

yes

[user] ?- member(a,[]) # member(a,[a]).<ENTER>

yes
```

The evaluation of complex goals becomes critical if operators with the same precedence and different meanings appear together. The operator for the OR function of Prolog is defined as `op(1100,xfy,;)`. If the database contains three simple facts `a, b` and `c`, evaluation is performed as follows:

```
[user] ?- a ; b # c.<ENTER>

yes
```

This is because IF/Prolog inserts parentheses since both operators are right-associative, namely: `a ; (b # c)`.
The operators should now be redefined. The right-associative operators `#` and `;` are redefined as left-associative operators:

```
[user] ?- op(0,xfy,#).<ENTER>

yes

[user] ?- op(1100,yfx,#).<ENTER>

yes

[user] ?- op(0,xfy,;).<ENTER>

yes
```

```
[user] ?- op(1100,yfx,;).<ENTER>

yes

[user] ?- a ; b # c.<ENTER>

no
```

The parentheses are now inserted "from the left", namely `(a ; b) # c`. Evaluation changes with the associativity.

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *Priority* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

current_op/3

Define and delete operators (directive)

:- op(+*Priority*, +*Assoc*, @*Names* **)**

The directive `op/3` can be used to define operators and delete operator definitions. It is analogous to the predicate `op/3`. As as directive, it is executed by `consult/1`, `reconsult/1` or `compile/1` when a Prolog source text is read and takes effect immediately.

Arguments

Priority	Integer or integer expression $0 \leq Priority \leq 1200$
Assoc	Atom: xf | yf | fx | fy | xfx | xfy | yfx
Names	Atom or list of atoms

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
: The argument *Priority*, *Assoc* or *Names* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
: The argument *Priority* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(operator_priority)
: The argument *Priority* is not in the prescribed range of values for the precedence of operators.

type_error(atom)
: The argument *Assoc* or an element of *Names* must be an atom, but is a term of another type.

domain_error(operator_specifier)
: The argument *Assoc* is not a valid type for an operator definition.

type_error(list)
: The argument *Names* must be a regular list or an atom, but is a term or another type or not regular.

permission_error(modify,operator)
The argument *Names* specifies an existing operator, whose definition is incompatible with the new definition.

permission_error(create,operator)
The argument *Names* specifies an existing operator that may not be redefined.

Hints

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Operator declarations take effect as soon as the directive is evaluated.

Standard

The directive `op/3` complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *Priority* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

op/3, current_op/3

Open an input/output stream

open(@*DeviceAndName*, +*Mode*, -*Stream* **)**

open(@*DeviceAndName*, +*Mode*, -*Stream*, @*Options* **)**

The predicate `open/4` opens *DeviceAndName* for input or output in accordance with the specified *Mode* and *Options*.

If *DeviceAndName* is an atom, a file is opened with this name. Otherwise, *DeviceAndName* must be a structure of the form *Device*`(`*Name*`)`, where *Device* is the name of an installed device driver (see `stream_device/2`) and *Name* is an atom that is used by the device driver for identification purposes.

Stream must be an uninstantiated variable. It is instantiated to a system-generated term.

Mode specifies the I/O direction for which the stream is to be opened. The following entries are possible:

`read` The stream is opened for reading.

`write` The stream is opened for writing. If the *DeviceAndName* object to be opened (e.g. a file) already exists, the original contents are overwritten.

`append` The stream is opened for writing. If the *DeviceAndName* object to be opened (e.g. a file) already exists, the stream is appended to the original contents of the object without overwriting.

Options must be specified in the form of a list which may contain the following elements:

`type`(*Type*)
: This specification defines the type of stream. The following entries are possible for *Type*:

  `text` For a stream containing characters (default).

  `binary` For a stream containing binary data

  The `text` and `binary` types are equivalent under UNIX. Other operating systems may involve specific conversions when certain characters are read/written.

`buffer`(*Buffering*)
: Defines how data is to be buffered by IF/Prolog. The following entries are possible for *Buffering*:

  `no` No buffering

  `line` Line buffering (only meaningful with the `text` option).

`full` Device-dependent buffering (default).

`reposition`(*Bool*)
Enables positioning and subsequent repositioning within the stream. The following entries are possible for *Bool*:

`true` Repositioning shall be possible.

`false` Repositioning need not be possible.

`alias`(*Alias*)
Assigns the specified *Alias* to the stream after it is opened (see `assign_alias/2`).

`eof_action`(*Action*)
Defines how IF/Prolog will respond if an attempt is made to execute further read operations after the end of the input stream is reached. This entry is only possible for input streams and does not take effect until the end of the stream has been normally reached at least once. The following values are possible for *Action*:

`error` Generates an exception if additional read operations are performed after the end of the input stream is reached (default, except for terminals).

`eof_code`
All reading predicates behave as if the end of the input stream was reached for the first time and return the indicator intended for that purpose (e.g. `end_of_file` with `get_char/1/2` or `-1` with `get_code/1/2`).

`reset` If an attempt is made to read a stream again after the end of the input stream has been reached, the stream is *reset* (i.e. returned to the state before the end of stream was reached for the first time), and the read operation is repeated. The actual effect of *resetting* an input stream in specific cases is defined by the respective device driver. This entry should therefore only be used if the corresponding effect is known and also makes sense (default behavior for terminals).

The predicate `open/3` is identical to `open/4` with default options.

Arguments

DeviceAndName	Atom or structure: name of the input/output medium to be opened.
Mode	Atom: `read | write | append`
Options	List of I/O options
Stream	Stream (system-defined term)

Exceptions

instantiation_error
The argument *DeviceAndName*, *Mode* or *Options* or a subterm must not be a variable, but a variable was specified.

domain_error(source_sink)
The value of the argument *DeviceAndName* is not valid for an object to be opened.

type_error(atom)
The argument *Mode* must be an atom, but is a term of another type.

domain_error(io_mode)
The value of the argument *Mode* is invalid for the I/O mode of a stream.

type_error(list)
The argument *Options* must be a list, but is a term of another type.

domain_error(stream_option)
An element of *Options* specifies an option that is not permitted.

type_error(variable)
The argument *Stream* must be a variable, but is a term of another type.

permission_error(open,source_sink)
The argument *DeviceAndName* specifies a file (or a different I/O medium) that cannot be opened.

existence_error(open,source_sink)
The argument *DeviceAndName* specifies a file (or a different I/O medium) that cannot be opened for reading.

domain_error(file_name)
The value of the argument *DeviceAndName* is not a valid name for a file.

permission_error(open,source_sink)
An element of *Options* specifies an alias that is already associated with an open stream.

permission_error(open,source_sink)
An element of *Options* is `reposition(true)`, but it is not possible to reposition this stream.

Hints

With access mode `read`, the Prolog flag `search_path` will be used as search path for files to be opened (see `set_prolog_flag/2`).

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The option `buffer(...)` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

close/1/2, stream_device/2, stream_control/2, stream_property/2

Position the cursor

outtab(+*Position* **)**

outtab(@*Stream*, +*Position* **)**

The predicate `outtab/1` sets the cursor in the current output stream (`current_output`) to the specified *Position* on the current line, provided the cursor was previously to the left of *Position*. Otherwise, the cursor is moved to the specified *Position* on the next line.

The predicate `outtab/2` has the same effect, but on the specified output stream *Stream*.

Arguments

Position	Integer or integer expression
Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* or *Position* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Position* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Furthermore, if an arithmetic expression is specified for the argument *Position*, all the exceptions for `is/2` can occur.

Hints

If the output is sent to a terminal, and inputs are made at the same time, the results may be unexpected, since IF/Prolog cannot take inputs into account when calculating the position in an output stream.

The predicate `outtab/1/2` positions the cursor absolutely; `tab/1/2` positions it relative to its current position.

Example

Tabulated output of visible predicates:

```
[user] ?- nl, current_visible(Mod,Pred/Arity),<ENTER>
          write(Mod), outtab(10),<ENTER>
          write(Pred), outtab(30),<ENTER>
          write(Arity), nl,<ENTER>
          Pred = system.<ENTER>

system   .                  2
system   =                  2
system   calling_context    1
system   catch              3
system   is                 2
system   <                  2
system   =<                 2
system   >=                 2
system   >                  2
system   =:=                2
system   =\=                2
system   system             5

Mod     = system
Pred    = system
Arity   = 5 <ENTER>

yes

[user] ?- write('First line'), outtab(10),<ENTER>
          write('Table stop'), nl.<ENTER>
First line
         Table stop

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

tab/1/2, write/1/2

Parse a character string in accordance with Prolog syntax

parse_atom(+*String*, +*StartPosition*, ?*EndPosition*, ?*Term*, ?*VarList*, ?*Error* **)**

The predicate `parse_atom/6` parses *String* in accordance with the Prolog syntax (see section 9). Parsing starts at *StartPosition* and ends at the first period (`.`) which cannot be part of the term syntax. This may be:

- a period followed by a delimiter, or
- a period at the end of *String*.

If a correct term is recognized, this term is unified with *Term*, and *Error* is unified with 0. In addition, *VarList* is unified with the list of variables and their names (as atoms) found in *Term* in the form:

`[` *Name* `=` *Variable*`, ... ]`

If *String* starting at *StartPosition* is not a correct Prolog term, then *Term* and *VarList* are not unified, i.e. remain uninstantiated when variables are specified, and *Error* is unified with the appropriate syntax error number (see `syntax_error/2`). The Prolog flag `report_read_error` (see `set_prolog_flag/2`) can be used to control whether or not a syntax error message is to be output.

EndPosition is always unified with the integer that specifies the position in *String* at which parsing ends so that further `parse_atom/6` calls, if any, can be continued from this position.

Arguments

String	Atom
StartPosition	Integer or integer expression $0 <$ *StartPosition* $\leq$ Length of *String*
EndPosition	Integer
Term	Term
VarList	List
Error	Integer ≥ 0

Exceptions

instantiation_error
: The argument *String* or *StartPosition* must not be a variable, but a variable was specified.

type_error(atom)
: The argument *String* must be an atom, but is a term of another type.

type_error(integer)
The argument *StartPosition* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(atom_length)
The argument *StartPosition* is not in the prescribed range of values.

type_error(integer)
The argument *EndPosition* or *Error* must be a variable or an integer, but is a term of another type.

type_error(list)
The argument *VarList* must be a variable or a list, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *StartPosition*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- parse_atom('abc.',1,P,T,V,E).<ENTER>

P       = 5
T       = abc
V       = []
E       = 0 <ENTER>

yes

[user] ?- parse_atom('a c.',1,P,T,V,E).<ENTER>
*** syntax_error: operator expected
>>> line 1, column 3

P       = 5
T       = _69
V       = _70
E       = 7 <ENTER>

yes
```

The first period in the string is the decimal point of a floating-point number. Parsing therefore ends with the second period.

```
[user] ?- parse_atom('A+1.3.',1,P,T,V,E).<ENTER>

P       = 7
T       = _134 + 1.3
V       = ['A' = _134]
E       = 0 <ENTER>
```

```
yes
```

More then one term from a string can be parsed:

```
[user] ?- Line_of_Terms = 'a(A). b(B).',<ENTER>
          parse_atom(Line_of_Terms,1,P,Y,YY,_),<ENTER>
          parse_atom(Line_of_Terms,P,_,Z,ZZ,_).<ENTER>

Line_of_Terms = 'a(A). b(B).'
P       = 7
Y       = a(_210)
YY      = ['A' = _210]
Z       = b(_218)
ZZ      = ['B' = _218] <ENTER>

yes
```

A period may be part of an atom which is made up of special characters There must therefore be a delimiter between a special character atom and the terminating period.

```
[user] ?- parse_atom( '## .',1,P,T,V,E).<ENTER>

P       = 5
T       = ##
V       = []
E       = 0 <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

read_error/2, syntax_error/2, set_prolog_flag/2, current_prolog_flag/2, atom_length/2

Input a byte

peek_byte(?*ByteCode*)

peek_byte(@*Stream*, ?*ByteCode*)

The predicate `peek_byte/1` peeks at the next byte from the *current* input stream (`current_input`) and unifies *ByteCode* with its integer value.

The predicate `peek_byte/2` acts in the same way for the specified input stream *Stream*.

If the end of the input stream has been reached, *ByteCode* is unified with `-1`.

The predicates can be applied only for binary streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
ByteCode	Integer, $-1 \leq ByteCode \leq 255$

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,text_stream)
The current input stream or the argument *Stream* must be a binary stream, but is associated with a text stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

type_error(integer)
The argument *ByteCode* must be a variable or an integer, but is a term of another type.

representation_error(in_byte)
The argument *ByteCode* must be a variable or an integer in the range -1 $\leq$ *ByteCode* $\leq$ 255.

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(input,text_stream)` is generated instead of `permission_error(input,binary_stream)`, when the stream is not a binary stream.

See also

get_byte/1/2, peek_char/1/2, peek_code/1/2

Input a character

peek_char(?*Character*)

peek_char(@*Stream*, ?*Character*)

The predicate `peek_char/1` peeks at the next character from the *current* input stream (`current_input`) and unifies *Character* with the character read in.

The predicate `peek_char/2` acts in the same way for the specified input stream *Stream*.

If the end of the input stream has been reached, *Character* is unified with the atom `end_of_file`.

The predicates can be applied only for text streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
Character	Character or `end_of_file`

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,binary_stream)
The current input stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)

system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

type_error(atom)
The argument *Character* must be a variable or an atom, but is a term of another type.

representation_error(in_character)
The argument *Character* must be a variable, a character or the special atom `end_of_file`, but is not representable as a single character.

representation_error(character)
The entity read from the stream is not representable as a single character.

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(input,binary_stream)` is generated instead of `permission_error(input,text_stream)`, when the stream is not a text stream.

See also

get_char/1/2, peek_byte/1/2, peek_code/1/2

Input a character

peek_code(?*CharCode*)

peek_code(@*Stream*, ?*CharCode*)

The predicate `peek_code/1` peeks at the next character from the *current* input stream (`current_input`) and unifies *CharCode* with its integer value.

The predicate `peek_code/2` acts in the same way for the specified input stream *Stream*.

If the end of the input stream has been reached, *CharCode* is unified with `-1`.

The predicates can be applied only for text streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
CharCode	Integer

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,binary_stream)
The current input stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

type_error(integer)
The argument *CharCode* must be a variable or an integer, but is a term of another type.

representation_error(in_character_code)
The argument *CharCode* must be a variable, an integer representing a character code or the integer -1.

representation_error(character)
The entity read from the stream is not representable as a single character.

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(input,binary_stream)` is generated instead of `permission_error(input,text_stream)`, when the stream is not a text stream.

See also

get_code/1/2, peek_byte/1/2, peek_char/1/2

Call a grammar rule

phrase(+*Goal*, ?*List*)

The predicate `phrase/2` can be used to call a goal that has been generated from grammar rules. The grammar rules are generated with the language construct `-->/2`.

The argument *Goal* has been given as the left side of a grammar rule. The argument *List* is unified by backtracking with all productions generated with the rule.

Arguments

Goal	Term
List	List

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

Example

The following grammar rules have been entered into the database:

```
sentence --> nom_phrase, verb_phrase.
nom_phrase --> [dogs].
verb_phrase --> [bark].
```

The first grammar rule can be called as follows:

```
[user] ?- phrase(sentence, L).

L       = [dogs,bark] <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

`-->`/2, expand_term/2

Query global variable and remove value

pop_global(+*Name*, ?*Value* **)** `[` **@** +*Module* `]`

The predicate `pop_global/2` unifies *Value* with the topmost element of the value stack of the global variable *Name* in the calling module or in the specified *Module*, and removes this value from the stack.

If the value stack of the global variable *Name* is emptied as a result, the global variable is deleted (`unset_global/1`).

Arguments

Name	Atom, name of a global variable
Value	Term

Exceptions

instantiation_error
The argument *Name* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

existence_error(global_variable)
The argument *Name* is not the name of an existing global variable.

Hints

Global variables are always assigned to a module; they can be neither exported nor imported. In operations with global variables, therefore, only those global variables defined in the calling module or in the specified *Module* are taken into account.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_global/1, get_global/2, push_global/2, set_global/2, unset_global/1

User-defined output predicate

portray(@*Stream*, @*Term* **)** `[ :-` *Body* `]`

The predicate `portray/2` enables the user to influence the output of terms.

The predicate `portray/2` is called by `print/1/2` and possibly by `write_term/2/3` if it is defined or visible in the module in which the original output predicate is executed.

If the predicate `portray/2` fails or if an exception occurs during execution, the term is output as if `portray/2` were undefined. In the case of structures, `portray/2` is also called for the output of subterms if `portray/2` could not be successfully executed for the full term. This allows the user to control the output of specific terms.

Arguments

Stream	Stream (system-defined term)
Term	Term

Hints

The predicate `portray/2` is always called with a stream argument, even if `print/1` or `write_term/2` was called, or `print/2` or `write_term/3` was called using an alias.

Example

The empty list is to be printed in a special format:

```
[user] ?- listing.<ENTER>

% *** user: portray / 2 ***
portray(Stream,[]) :-
    write(Stream,'[/*empty list*/]') .

yes

[user] ?- print([]),nl.<ENTER>
[/*empty List*/]

yes

[user] ?- print(a(b,[])),nl.<ENTER>
a(b,[/*empty List*/])

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

print/1/2, write_term/2/3

Query and set the exception mode of a predicate

predicate_mode(@*Predicate*, ?*Before*, +*After* **)** `[` **@** +*Module* `]`

The predicate `predicate_mode/3` defines for *Predicate* whether the exception `existence_error(procedure,...)` is to be generated if this predicate is not yet or no longer defined (see `abolish/1`) at the time it is called.

Identification of the predicate is based on its visibility in the calling module or in the specified *Module*.

The argument *Before* is unified with the previously valid mode. The exception mode is then set in accordance with the *After* specification.

If *After* is set to

`on` no exception will be generated if *Predicate* is called and is not (or no longer) defined. The call simply fails.

`off` the exception `existence_error(procedure,...)` will be generated if *Predicate* is called and not (or no longer) defined.

The *Predicate* need not yet be defined. This attribute is retained beyond `abolish/1`.

Arguments

Predicate	Structure of the form *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$
Before	Atom: on | off
After	Atom: on | off

Exceptions

instantiation_error
The argument *Predicate* or *After* or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
The argument *Predicate* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

type_error(atom)
The argument *After* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *After* must be one of the atoms `on` or `off`, but is another atom.

type_error(atom)
The argument *Before* must be a variable or an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Before* must be a variable or one of the atoms `on` or `off`, but is another atom.

permission_error(access,system_module)
The predicate `predicate_mode/3` must not be executed for a system module.

Hints

The exception mode is normally set to `on` only for predicates that were declared with `dynamic/1`.

The predicate `predicate_mode/3` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

A predicate may have certain attributes, which, when it is created, are set implicitly by the system according to the current environment or declarations. They can be modified to a certain extent explicitly by the user. These include the following:

- List mode (see `list_mode/3`)
- Debug mode (see `debug_mode/3`)
- Exception mode (see `predicate_mode/3`)
- Modification mode (see `modify_mode/3`)

These attributes are retained even if the predicate is deleted (with `abolish/1` for example).

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

abolish/1, dynamic/1, modify_mode/3

Query predicate type

predicate_type(@*Predicate*, ?*Type* **)** `[` @ +*Module* `]`

The predicate `predicate_type/2` determines the type of the *Predicate*. *Type* is unified with an atom that represents the type of the predicate.

Identification of the predicate is based on its visibility in the calling module or in the specified *Module*.

The following types are possible:

Type	Explanation
`builtin`	The predicate is defined and is either a built-in predicate or a C-language predicate.
`compiled`	The predicate is a user-defined and compiled Prolog predicate.
`imported`	The predicate is an imported predicate that is defined in another module.
`linear`	The predicate is a user-defined but uncompiled Prolog predicate or was declared as *dynamic*.
`control`	The argument is a language construct, not a predicate.
`undefined`	Otherwise

Arguments

Predicate	Structure of the form *Functor*/*Arity*
Type	Atom
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Predicate* or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
The argument *Predicate* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

type_error(atom)
The argument *Type* must be a variable or an atom, but is a term of another type.

Hints

Although language constructs (such as `->/2` etc.) are not predicates, the type `control` is returned for them in a type query so that they can be analyzed.

The predicate `predicate_type/2` is a metapredicate and considers only predicate definitions which are visible in the calling module or in the specified *Module*.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_predicate/1, current_visible/2

Output a term

print(@*Term* **)** `[` **@** @*Module* `]`

print(@*Stream*, @*Term* **)** `[` **@** +*Module* `]`

The predicates `print/1/2` attempt to call the user-defined predicate `portray/2` in the calling module or in the specified *Module* to output *Term*, if *Term* is not a variable.

If the call does not succeed, e.g. because `portray/2` is undefined, fails or returns an exception, the output is the same as with `write/1/2`. If *Term* is a structure, the attempt to call `portray/2` is repeated for each subterm in *Term*.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Term	Term

Exceptions

instantiation_error
: The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
: The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
: The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
: The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
: The indicated error was reported by the operating system when writing on the output stream *Stream*.

Hints

The call

```
print(Term)
```

is equivalent to

```
write_term(Term,[portrayed(true),numbervars(true)]) .
```

Example

Output of all integers as floating-point numbers:

```
[user] ?- listing.<ENTER>

% *** user: portray / 2 ***
portray(Stream,Number) :-
    integer(Number),
    Floating point is float(Number),
    writeq(Stream,Floating point) .

yes

[user] ?- print(3),nl.<ENTER>
3.0

yes

[user] ?- print([1,2]),nl.<ENTER>
[1.0,2.0]

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

portray/2, write/1/2, writeq/1/2, write_canonical/1/2, write_term/2/3

Declare predicates as private

:- private @*PredicateList*
:- private(@*PredicateList* **)**

The directive `private/1` is used to declare predicates that are to be defined in the current module.

This directive must be specified only if a predicate of the same name and same arity was imported by importing a module or if such a predicate is visible in the current module as a result of the implicit importing of system modules.

Since this directive is executed implicitly by the directives `export/1`, `meta/1` and `dynamic/1`, it need not be specified again for the predicates declared with these directives.

The directive `private/1` may appear only in the declaration part of a module body.

Arguments

PredicateList	List with the format `[`*Predicate*,...`]` or structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

permission_error(directive,module)
The specified directive `private/1` is not valid in this position.

Hints

The atom `private` is defined as a prefix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `private/1` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, import/1/2, meta/1, export/1, dynamic/1

Activate profiling for a goal

profile(+*Goal* **)** `[` **@** +*Module* `]`

profile(+*Goal*, @*Ports* **)** `[` **@** +*Module* `]`

The predicates `profile/1/2` activate the profiler for executing *Goal*.

The profiler counts on a pre predicate basis how many times Prolog execution passes given ports. The argument *Ports* specifies which ports should be counted.
The predicate `profile/1` is equal to the predicate `profile/2` with *Ports* set to the list `[ call,exit,redo,fail ]`.

! The profiler is based on the Prolog tracer. It (re-)defines the predicate `trace_output/4` in the module `user`. Therefore, the profiler should not be used in conjunction with the tracer.

Arguments

Goal	Goal
Ports	List of atoms: call | exit | fail | redo | suspend | wakeup | error | failmatch | enterbody | exitbody | redobody | trymatch | failbody

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

instantiation_error
The argument *Ports* must not be a variable, but a variable was specified.

type_error(list)
The argument *Ports* must be a list, but is a term of another type.

type_error(atom)
An element of *Ports* must be an atom, but is a term of another type.

Hints

The profiling predicates are included in the module `profiler`. This module must be imported before the predicates can be used.

These predicates are not part of the IF/Prolog runtime system and are therefore not available for applications intended for third parties.

The predicate `profile/1/2` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicates `profile/1/2` are new.

See also

profile_stat/0/1, profile_reset/0, trace/1, trace_output/4, trace_config/3, debug/1, call/1, set_prolog_flag/2, import/1

Reset profiling statistics

profile_reset

The predicate `profile_reset/0` resets the profiling data.

Hints

The profiling predicates are included in the module `profiler`. This module must be imported before the predicates can be used.

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `profile_reset/0` is new.

See also

profile/1/2, profile_stat/0/1, trace/1, trace_output/4, trace_config/3, debug/1, call/1, set_prolog_flag/2, import/1

Query profiling statistics

profile_stat

profile_stat(?*List*)

The predicate `profile_stat/1` queries the profiling data. The argument *List* is unified with an unsorted list of compound terms of the form

`profile_data`(*Predicate*, *Port*, *Count*)

where *Predicate* is a predicate indicator, *Port* is an atom and *Count* is a positive integer.

The predicate `profile_stat/0` outputs the profiling statistics (sorted by predicate name) to the current output stream.

Arguments

List	List
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$
Port	Atom: call | exit | fail | redo | suspend | wakeup | error | failmatch | enterbody | exitbody | redobody | trymatch | failbody
Count	Integer

Exceptions

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

Hints

The profiling predicates are included in the module `profiler`. This module must be imported before the predicates can be used.

These predicates are not part of the IF/Prolog runtime system and are therefore not available for applications intended for third parties.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicates `profile_stat/0/1` are new.

See also

profile/1/2, profile_reset/0, trace/1, trace_output/4, trace_config/3, debug/1, call/1, set_prolog_flag/2, import/1

Automatically call a Prolog goal

program [:- *Body*]

The predicate `program/0` in the module `user` is not a built-in predicate but one which can be defined by the user. It is used to ensure that a Prolog application runs automatically.

If, after IF/Prolog has been started and the system parameters have been processed, i.e. when all load and consult requests given on the command line have been executed, the predicate `program/0` is visible in the module `user`, then this predicate is called instead of starting the Prolog input loop (`break/0`). Once it has been executed, IF/Prolog is exited.

Hints

Any application that is activated by `program/0` must be fully and independently responsible for error handling. Since the context which is otherwise defined in the Prolog input loop to trap all exceptions is not active at the time `program/0` is initiated, any exception that occurs when executing `program/0` will result in the termination of the program, unless such exceptions are handled or trapped in the application itself.

Example

The file *example.pro* contains the following application:

```
program :- test.
program :- write(ende),nl.

test :- write(test),nl,fail.
```

If this file is loaded on invoking IF/Prolog, the application will be executed immediately.

```
$ prolog -nonotify -c example.pro<ENTER>
IF/PROLOG V5.1
Copyright (C) Siemens Nixdorf Information Systems AG, 1996

test
ende
$
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

break/0, system_parameters/1

Query parameters from IF/Prolog call

program_parameters(?*ParameterList*)

The predicate `program_parameters/1` unifies *ParameterList* with a list of all parameters which were specified on calling IF/Prolog.

The first element of the list is the name under which IF/Prolog was called.

Arguments

ParameterList	List of atoms

Exceptions

type_error(list)
The argument *ParameterList* must be a variable or a list, but is a term of another type.

type_error(atom)
An element of *ParameterList* must be a variable or an atom, but is a term of another type.

Hints

The first element of *ParameterList* need not match the name given by the user on the command line; it may also be an appropriately expanded pathname under which IF/Prolog was found in the system. This name will usually vary from system to system.

Example

The file *example.pro* contains the following Prolog program:

```
program :-
        program_parameters(List),
        write(List),nl.
```

Calling IF/Prolog as shown below would then return:

```
$ prolog -nonotify -c example -data hello<ENTER>
IF/PROLOG V5.1
Copyright (C) Siemens Nixdorf Information Systems AG, 1996

[prolog,-nonotify,-c,example,-data,hello]
$
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

system_parameters/1, user_parameters/1

Query IF/Prolog serial number

prolog_serial(?*Serial*)

The predicate `prolog_serial/1` unifies *Serial* with an atom which contains the serial number of IF/Prolog.

The serial number varies from one operating system to another and changes with each new release of IF/Prolog. It can be used to control system- and version-specific actions in an application.

Arguments

Serial	Atom

Exceptions

type_error(atom)
The argument *Serial* must be a variable or an atom, but is a term of another type.

Example

```
[user] ?- prolog_serial(Serial).<ENTER>

Serial  = 'kb-511000000' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `prolog_serial/1` is new.

See also

system_hostid/1, system_name/1, prolog_version/1, proroot/1

Query IF/Prolog version

prolog_version(?*Version*)

The predicate `prolog_version/1` unifies *Version* with an atom which contains the version number of IF/Prolog.

The version number changes with each new release of IF/Prolog and can be used to control version-specific actions in an application.

Arguments

Version	Atom

Exceptions

type_error(atom)
The argument *Version* must be a variable or an atom, but is a term of another type.

Example

```
[user] ?- prolog_version(Ver).<ENTER>

Ver     = '5.1A' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

prolog_serial/1, proroot/1, system_hostid/1, system_name/1

Query path of IF/Prolog system

proroot(?*Path* **)**

The predicate `proroot/1` unifies *Path* with an atom containing the installation path of IF/Prolog in the current operating system.

The installation path will usually vary from system to system.

The installation path can be made known to IF/Prolog explicitly using the system parameter `-root` or the environment variable `PROROOT`.

Arguments

Path Atom

Exceptions

type_error(atom)
The argument *Path* must be a variable or an atom, but is a term of another type.

Example

```
[user] ?- proroot(Root).<ENTER>

Root    = '/usr2/ifprolog/system' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

system_name/1, prolog_version/1

Set value of a global variable

push_global(+*Name*, @*Value* **)** [@ +*Module*]

The predicate `push_global/2` pushes *Value* onto the value stack of the global variable *Name* in the calling module or in the specified *Module*, thus making *Value* the new topmost element of the stack.

If the global variable *Name* does not exist, it is created.

Arguments

Name	Atom
Value	Term

Exceptions

instantiation_error
The argument *Name* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

Hints

Global variables are always assigned to a module; they can be neither exported nor imported. In operations with global variables, therefore, only those global variables defined in the calling module or in the specified *Module* are taken into account.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_global/1, get_global/2, pop_global/2, set_global/2, unset_global/1

Output a byte

put_byte(+*ByteCode* **)**

put_byte(@*Stream*, +*ByteCode* **)**

The predicate `put_byte/1` outputs the byte *ByteCode* to the *current* output stream (`current_output`).

The predicate `put_byte/2` has the same effect, but on the specified output stream *Stream*.

The predicates can be applied only for binary streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
ByteCode	Integer or integer expression, $0 \leq ByteCode \leq 255$

Exceptions

instantiation_error
: The argument *Stream* or *ByteCode* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
: The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
: The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
: The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

permission_error(output,text_stream)
: The current output stream or the argument *Stream* must be a binary stream, but is associated with a text stream.

type_error(integer)
: The argument *ByteCode* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

representation_error(byte)
: The argument *ByteCode* must be an integer in the range $0 < ByteCode \leq 255$.

system_error(write)
system_error(close)

system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Furthermore, if an arithmetic expression is specified for the argument *ByteCode*, all the exceptions for `is/2` can occur.

Example

Output of ASCII characters:

```
[user] ?- put_code(97), put_code(98), put_code(10).<ENTER>
ab

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *ByteCode* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

Compatibility

V5.0B ISO conformance: exception `permission_error(output,text_stream)` is generated instead of `permission_error(output,binary_stream)`, when the stream is not a binary stream.

See also

get_byte/1/1, put_char/1/2, put_code/1/2

Output a character

put_char(+*Char* **)**

put_char(@*Stream*, +*Char* **)**

The predicate `put_char/1` outputs a character *Char* to the *current* output stream (`current_output`).

The predicate `put_char/2` has the same effect, but on the specified output stream *Stream*.

The predicates can be applied only for text streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
Char	Atom, character

Exceptions

instantiation_error
The argument *Stream* or *Char* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

permission_error(output,binary_stream)
The current output stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

type_error(character)
The argument *Char* must be a character, but is a term of another type.

representation_error(character)
The argument *Char* is not representable as a single character.

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Example

Output of individual characters:

```
[user] ?- put_char(a), put_char('\t'),<ENTER>
          put_char(b), put_char('\n').<ENTER>
a        b

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(output,binary_stream)` is generated instead of `permission_error(output,text_stream)`, when the stream is not a text stream.

See also

get_char/1/2, put_byte/1/2, put_code/1/2

Output a character

put_code(+*CharCode* **)**

put_code(@*Stream*, +*CharCode* **)**

The predicate `put_code/1` outputs a character which has the character code *CharCode* to the *current* output stream (`current_output`).

The predicate `put_code/2` has the same effect, but on the specified output stream *Stream*.

The predicates can be applied only for text streams (see `open/3/4`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
CharCode	Integer or integer expression

Exceptions

instantiation_error
The argument *Stream* or *CharCode* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

permission_error(output,binary_stream)
The current output stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

type_error(integer)
The argument *CharCode* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

representation_error(character_code)
The argument *CharCode* must be an integer representing a character code.

system_error(write)
system_error(close)

system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Furthermore, if an arithmetic expression is specified for the argument *CharCode*, all the exceptions for `is/2` can occur.

Example

Output of ASCII characters:

```
[user] ?- put_code(97), put_code(98), put_code(10).<ENTER>
ab

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the argument *CharCode* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for this predicate. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

Compatibility

V5.0B ISO conformance: exception `permission_error(output,binary_stream)` is generated instead of `permission_error(output,text_stream)`, when the stream is not a text stream.

See also

get_code/1/1, put_byte/1/2, put_char/1/2

Test for rational number

rational(@*TestTerm* **)**

The predicate `rational/1` succeeds if *TestTerm* is a rational number (but not an integer), otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- rational(0r3/7).<ENTER>

yes

[user] ?- X is 0r1/6 + 0r1/5 + 0r1/4 + 0r1/3 + 0r1/2,<ENTER>
          rational(X).<ENTER>

X       = 0r29/20 <ENTER>

yes

[user] ?- rational(7).<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

atom/1, atomic/1, compound/1, cyclic/1, float/1, ground/1, integer/1, nonvar/1, number/1, var/1

Decompose a rational number

rational(+*Number*, ?*Numerator*, ?*Denominator* **)**

The predicate `rational/3` determines the denominator and numerator of a rational number.

If *Number* is a rational number, then *Numerator* is unified with the numerator and *Denominator* with the denominator of this number. If *Number* is an integer, *Numerator* is unified with *Number* and *Denominator* with 1.

Arguments

Number	Rational number or integer
Numerator	Integer
Denominator	Integer

Exceptions

instantiation_error
The argument *Number* must not be a variable, but a variable was specified.

type_error(rational)
The argument *Number* must be a rational number or an integer, but is a term of another type.

type_error(integer)
The argument *Numerator* or *Denominator* must be a variable or an integer, but is a term of another type.

Example

```
[user] ?- rational(0r3/7, Numer, Denom).<ENTER>

Numer   = 3
Denom   = 7 <ENTER>

yes

[user] ?- X is 0r1/6 + 0r1/5 + 0r1/4 + 0r1/3 + 0r1/2,<ENTER>
          rational(X, Numer, Denom).<ENTER>

X       = 0r29/20
Numer   = 29
Denom   = 20 <ENTER>
```

```
yes

[user] ?- rational(7, Numer, Denom).<ENTER>

Numer   = 7
Denom   = 1 <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

rational/1, is/2 (operator `rdiv`)

Input a term

read(?*Term* **)**

read(@*Stream*, ?*Term* **)**

The predicate `read/1` reads in a term from the current input stream and unifies it with *Term*.

The predicate `read/2` acts in the same way as `read/1`, but reads from the specified input stream *Stream*.

If the end of the input stream is reached, *Term* is unified with the atom `end_of_file`.

If a syntax error is detected when attempting to read in a valid Prolog term, the behavior of `read/1/2` is determined by the setting of the Prolog flag `read_error` (see `current_prolog_flag/2`).

Arguments

Stream	Stream (system-defined term) or alias (atom)
Term	Term

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,binary_stream)
The current input stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)

system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

syntax_error
Syntax errors were detected during analysis of input. All the syntax errors described for `syntax_error/2` can occur.

Hints

The call

```
read(Stream,Term)
```

is identical to

```
read_term(Stream,Term,[]) .
```

If the read process is interrupted by a signal, the predicate fails, even if the signal is handled.

Example

```
[user] ?- read(T).<ENTER>
a + 3 * f(x).<ENTER>

T       = a + 3 * f(x) <ENTER>

yes

[user] ?- read(T1),read(T2).<ENTER>
[a,b,c]. 'that\'s it'.<ENTER>

T1      = [a,b,c]
T2      = 'that''s it' <ENTER>

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

Compatibility

V5.0B ISO conformance: exception `permission_error(input,binary_stream)` is generated instead of `permission_error(input,text_stream)`, when the stream is not a text stream.

See also

read_term/2/3

Query position of a syntax error

read_error(?*Line*, ?*Number* **)**

read_error(?*Line*, ?*Column*, ?*Number* **)**

The predicate `read_error/2` determines the number of the input line in which a syntax error was detected by the parser of IF/Prolog in the last call to `read/1/2`, `read_term/2/3`, or `parse_atom/6`, and unifies the result with *Line*. The number of the corresponding syntax error message is unified with *Number*.

The predicate `read_error/3` additionally determines the character position of the error in the input line and unifies the result with *Column*.

Arguments

Line	Integer
Column	Integer
Number	Integer

Exceptions

type_error(integer)
The argument *Line*, *Column* or *Number* must be a variable or an integer, but is a term of another type.

Example

```
[user] ?- read_term(T,[syntax_errors(quiet)]);<ENTER>
          read_error(A,B,C),<ENTER>
          syntax_error(C,X).<ENTER>
atom( .<ENTER>

T       = _68
A       = 6
B       = 7
C       = 5
X       = 'start of term expected' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

read/1/2, read_term/2/3, parse_atom/6, syntax_error/2

Input a term

read_term(?*Term*, +*Options*)

read_term(@*Stream*, ?*Term*, +*Options*)

The predicate `read_term/2` reads in a term from the current input stream under the control of the specified *Options* and unifies it with *Term*.

The predicate `read_term/3` acts in the same way as `read_term/2`, but reads from the specified input stream *Stream*.

If the end of the input stream is reached, *Term* is unified with the atom `end_of_file`.

The following control options may be specified as elements of the *Options* list:

`syntax_errors( +`*Flag*` )`
: Determines the behavior of `read_term/2/3` when syntax errors occur. If this option is omitted, the value of the Prolog flag `read_error` (see `current_prolog_flag/2`) is used.
The following entries are possible for *Flag*:

`error`	An exception is raised.
`quiet`	No error is reported; `read_term/2/3` fails.
`fail`	An error message is output (no exception); `read_term/2/3` fails.

`variables( ?`*Vars*` )`
: If a term is read in successfully, *Vars* is unified with a list of all named variables found in *Term*. The variables are ordered according to their first occurrence from left to right.

`variable_names( ?`*VarList*` )`
: If a term is read in successfully, *VarList* is unified with a list of all named variables found in *Term* and their names in the form `[`*Name* = *Variable*, ... `]`, i.e. anonymous variables are not included in the list.

`varnames( ?`*VarList*` )`
: If a term is read in successfully, *VarList* is unified with a list of all variables found in *Term* and their names in the form `[`*Name* = *Variable*, ... `]`, i.e. anonymous variables are also included.

singletons(?*VarList*)
: If a term is read in successfully, *VarList* is unified with a list of all variables that occur only once in *Term* and their names in the form `[`*Name* = *Variable*, ... `]`. Anonymous variables are not included in this list.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Options	List of structures
Term	Term
Flag	Atom: error | quiet | fail
Vars	List of variables
VarList	List in the form [*Name* = *Variable*,...]
Name	Atom
Variable	Variable

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(input,binary_stream)
The current input stream or the argument *Stream* must be a text stream, but is associated with a binary stream.

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

instantiation_error
The argument *Options* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *Options* must be a list, but is a term of another type.

domain_error(read_option)
An element of *Options* specifies an option that is not permitted.

syntax_error
Syntax errors were detected during analysis of input. All the syntax errors described for `syntax_error/2` can occur.

Hints

If the read process is interrupted by a signal, the predicate fails, even if the signal is handled.

Example

```
[user] ?- read_term(T,[varnames(V)]).<ENTER>
a(X,Y).<ENTER>

T       = a(_131,_132)
V       = ['X' = _131,'Y' = _132] <ENTER>

yes

[user] ?- read_term(T,[varnames(V),singletons(S)]).<ENTER>
a(X,Y) :- write(Y).<ENTER>

T       = a(_150,_151) :- write(_151)
V       = ['X' = _150,'Y' = _151]
S       = ['X' = _150] <ENTER>

yes

[user] ?- read_term(T,[syntax_errors(error)]).<ENTER>
a(X Y).<ENTER>

*** E X C E P T I O N: syntax_error
>>> line = 4
>>> column = 5
>>> syntax_error = 'operator expected'
>>> goal = system : read_term(_119,[syntax_errors(error)])

no

[user] ?- read_term(T,[syntax_errors(fail)]).<ENTER>
a(.<ENTER>
*** syntax_error: start of term expected
>>> line 4, column 3

no
```

```
[user] ?- read_term(T,[syntax_errors(quiet)]).<ENTER>
d ## f.<ENTER>

no

[user] ?- read_term(T,[variable_names(V)]).<ENTER>
a([_,C]).<ENTER>

T       = a([_131,_132])
V       = ['C' = _132] <ENTER>

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

The options `syntax_errors(...)` and `varnames(...)` are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.0B ISO conformance: exception `permission_error(input,binary_stream)` is generated instead of `permission_error(input,text_stream)`, when the stream is not a text stream.

See also

read/1/2, read_error/2/3, syntax_error/2

Read in a Prolog file

reconsult(+*Filename* **)** [**@** +*Module*]
[**-** +*Filename*] [**@** +*Module*]

The predicate `reconsult/1` reads the file *Filename* into the database in the calling module or in the specified *Module*.

To enable Prolog text to be entered interactively as well, the atom `user` may also be specified for *Filename*. In this case, `reconsult/1` reads from the current input stream and displays a `'>'` prompt to indicate that a clause or a directive is expected and a `'|'` prompt to indicate that a clause or a directive that has already begun has not yet been completed. By entering `end_of_file` or pressing the appropriate key it is possible to terminate the consult process.

Clauses occurring in *Filename* are read in with `read_term/2`, checked for correct syntax and inserted with `assertz/1` or `assertz_with_names/2` into the database of the relevant module. Clauses with syntax errors are not inserted into the database.

If there are already clauses in the database for a predicate defined in *Filename*, they are deleted before the new clauses are inserted in the database.

If the Prolog flag `debug` is set to `on`, `assertz_with_names/2` is used for insertion in the database so that, for example, `listing/0/1` can be used to display the clauses of these predicates with their original variable names. Otherwise, `assertz/1` is used.

If grammar rules formulated with `-->/2` occur in *Filename*, they are interpreted by `reconsult/1` and converted into normal Prolog clauses.

Directives occurring in *Filename* are evaluated and executed according to their functionality. This may affect the further operation of `reconsult/1` (operator and module definitions for example). The following directives are available:

Directive	Description
`op/3`	Define and delete operators
`char_conversion/2`	Define a character conversion
`include/1`	Include a Prolog file in the source text
`ensure_loaded/1`	Insert a Prolog file in the source text once
`discontiguous/1`	Declare discontiguous predicate definition
`multifile/1`	Declare predicates as being scattered over several files
`init/1`	Initialization goal
`initialization/1`	Initialization goal
`dynamic/1`	Declare predicates as modifiable
`module/1`	Define a module
`begin_module/1`	Define the body of a module
`end_module/0/1`	Conclude the definition of a module
`import/1/2`	Import predicates
`export/1`	Export predicates of current module
`reexport/1`	Reexport predicates of imported module

`meta/1`	Declare metapredicates
`private/1`	Declare predicates as private

Arguments

Filename	Atom, name of a file or the atom `user`

Exceptions

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Filename*.

permission_error(modify,static_procedure)
When loading the file *Filename*, a predicate is to be defined which already exists and which may not be modified.

In addition, all the error messages and warnings from `read_term/2`, `assertz/1` and `assertz_with_names/2` and may occur. These do not cause the load process to be aborted. You should be aware that the database may not necessarily contain what you expect it to contain.

Errors in directives are also output as error messages on the error output stream. Unknown directives are reported by the error message `domain_error(directive)`.

In the event of errors in a module interface, the affected module is deleted after *Filename* has been processed.

If there are errors in the declaration part of a module body, the affected module body is deleted after *Filename* has been processed.

When initialization directives are executed (see `init/1`), any exception may arise. These aborts the execution of the initialization goal, but does not abort the load process.

In addition, warnings are output as a function of the Prolog flag `warnings` in certain situations to alert you to possible sources of errors.

Hints

If *Filename* does not contain the extension *.pro* and if a file of this name cannot be found, the extension *.pro* is appended and the resultant filename used.

The predicate `reconsult/1` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

If *Filename* contains module definitions, then database manipulation takes place in these modules. If not, it is implicitly assumed that the contents of *Filename* belong to the body of the calling module or of the specified *Module*. The difference compared with an explicitly specified `body/1` directive at the start of *Filename* is that the associated body of the module is not first deleted.

If *Filename* contains only module interfaces and/or bodies, then `consult/1` and `reconsult/1` are prefectly identical. The definition of a module interface of an existing module with the same name is first deleted, and the definition of a module body overwrites an existing module body.

The `init/1` directives in *Filename* are also executed in the context of the relevant module.

If a module in *Filename* contains several `init/1` directives, then the execution sequence is undefined.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

consult/1, ./2, `-->`/2

Reduce the depth of a structure

reduce(+*Level*, @*Term*, ?*ReducedTerm* **)**

The predicate `reduce/3` converts *Term* to a term with a maximum structure depth of *Level* and unifies the result with *ReducedTerm*.

Arguments

Level	Integer or integer expression
Term	Term
ReducedTerm	Term

Exceptions

instantiation_error
The argument *Level* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Level* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

Furthermore, if an arithmetic expression is specified for the argument *Level*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- reduce(1,a(b(c(y),c(yy)),d),T).<ENTER>

T       = a / 2 <ENTER>

yes

[user] ?- reduce(3,a(b(c(y),c(yy)),d),T).<ENTER>

T       = a(b(c / 1,c / 1),d) <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Reexport predicates

:- reexport +*Module*
:- reexport(+*Module* **)**

:- reexport @*PredicateList* **from** +*Module*
:- reexport(+*Module*, @*PredicateList* **)**

The directive `reexport/1` makes all the predicates exported/reexported from the module *Module* available for other modules.

The directive `reexport/2` makes all the predicates specified in *PredicateList* available for other modules.

The exported predicates must be made visible with `import/1/2`.

The directives `reexport/1/2` may only occur in the module interface.

Arguments

Module	Atom, name of a module
PredicateList	List with the format `[`*Predicate*,...`]` or structure with the format *Predicate*,...
Predicate	Structure with the format *Functor*/*Arity*
Functor	Atom
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *Module* or *PredicateList* or a subterm must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

existence_error(module)
The argument *Module* is not the name of an existing module.

type_error(list)
The argument *PredicateList* must be a regular list, but is a term of another type or not regular.

type_error(predicate_indicator)
The argument *Predicate* in *PredicateList* must be a predicate indicator, i.e. a structure with the format *Functor* / *Arity*, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* in *PredicateList* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* in *PredicateList* must be an integer.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* in *PredicateList* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* in *PredicateList* must be an integer in the range 0..127, but is larger.

permission_error(modify,procedure)
The predicate indicator *Predicate* in *PredicateList* specifies a predicate which already exists and is not modifiable or which is the name of a language construct.

existence_error(procedure)
The argument *Predicate* in *PredicateList* specifies a predicate that was not exported from the specified module.

permission_error(directive,module)
The specified directive `reexport/1/2` is not valid in this position.

Hints

The atom `reexport` is defined as a prefix operator and the atom `from` is defined as a infix operator.

Directives are evaluated during consultation (see `consult/1`, `reconsult/1`) or compilation (see `compile/1`, `procmp` command).

Standard

The directive `reexport/1/2` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, import/1/2, export/1, meta/1, private/1

Match a regular expression

regexp(+*RegExp*, +*Atom* **)**
regexp(+*RegExp*, +*Atom*, ?*List* **)**

The predicates `regexp/2/3` match *Atom* the regular expression *RegExp*. The predicates succeed when *Atom* or a subatom of *Atom* matches *RegExp*, otherwise they fail. With `regexp/3` the matching subatoms of *Atom* can be retrieved in *List*.

RegExp can be formed as follows:

`c` The character c where c is not a special character.

\c The character *c* where *c* is any character, except a digit in the range 1 - 9.

^ The beginning of the atom compared.

$ The end of the atom compared.

. Any character in the atom.

`[`*s*`]` Any character in the set *s*, where *s* is a character and/or a range, e.g. `[c-f]`.

`[^s]` Any character not in the set s, where s is defined as above.

`r*` Zero or more successive occurrences of the regular expression `r`. The longest match is chosen.

`rx` The occurrence of regular expression `r` followed by the occurrence of regular expression `x` (Concatenation).

r\{*m*, *n*\} Any number of m through n successive occurrences of the regular expression r. The regular expression r\{m\} matches exactly m occurrences, r\{m,\} matches at least m occurrences.

\(r\) The subatom matching the regular expression r is returned in the result list *List*.

Arguments

RegExp	Atom, a regular expression
Atom	Atom
List	List of atoms

Exceptions

type_error(atom)
The argument *RegExp* must be an atom, but is a term of another type.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(atom)
An element of *List* must be an atom, but is a term of another type.

domain_error(*Description***)**
The argument *RegExp* is not a legal regular expression. *Description* is a closer description of the error.

Hints

This predicate is supplied as an alternative for `match/2/3` to be used with more complex regular expressions. It works only for atoms made up of ASCII characters.

Example

```
[user] ?- regexp('\\([0-9]*\\)\\.\\([0-9]*\\)\\.\\([0-9]*\\)',
'1.12.1987', K).

K       = ['1','12','1987'] <ENTER>

yes

[user] ?- regexp('\\([0-9]\\{2\\}\\)', 'a1b12c123', K).

K       = ['12'] <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

match/2/3, match_atom/2

Remove a sublist from a list

remove(+*Sublist*, +*List*, ?*Remainder* **)**

The predicate `remove/3` removes from *List* the first sublist, whose elements can be unified sequentially with the elements of *Sublist*. The list with the remaining elements is unified with *Remainder*.

Arguments

Sublist	List
List	List
Remainder	List

Exceptions

instantiation_error
The argument *Sublist* or *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *Sublist* or *List* must be a list, but is a term of another type.

type_error(list)
The argument *Remainder* must be a variable or a list, but is a term of another type.

Example

Sublist is removed only once:

```
[user] ?- remove([a,b], [a,b,c,d,a,b], L).<ENTER>

L       = [c,d,a,b] <ENTER>

yes

[user] ?- remove([d,e], [a,b,c,d,e,f,g], L).<ENTER>

L       = [a,b,c,f,g] <ENTER>

yes
```

The elements of *Sublist* must be contiguous in *List*:

```
[user] ?- remove([a,b], [a,c,b], L).<ENTER>

no
```

The elements must be unifiable sequentially. Backtracking does not occur.

```
[user] ?- remove([A, B], [a,b,c,d], L).<ENTER>

A       = a
B       = b
L       = [c,d] ;<ENTER>

no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

append/3, member/2

Generate a choice point

repeat

The predicate `repeat/0` is always successful; in contrast to the predicate `true/0`, it is re-executable and will succeed any number of times by backtracking.

The predicate `repeat/0` is thus suitable for programming iterative loops in IF/Prolog by backtracking.

Hints

Variable instantiations which arise from attempts to satisfy a goal are undone again in the course of backtracking. Intermediate results obtained from a loop programmed with `repeat/0` are thus lost with each backtracking cycle.

When subgoals are combined with AND (`,/2`), subgoals to the left of a `repeat/0` subgoal are never reached again by backtracking.

If a predicate which always fails (e.g `fail/0`) is called after `repeat/0`, it may create an endless loop.

Loops created with `repeat/0` should always be terminated with a **cut** (`!/0`) to prevent the loop from being traversed again after the end-of-loop criterion has been reached.

Example

The following example reads terms and pushes them onto a stack:

```
[user] ?- repeat,<ENTER>
            read(Term),<ENTER>
            push_global(stack,Term),<ENTER>
          Term == end, !.<ENTER>
line1.<ENTER>
line2.<ENTER>
end.<ENTER>

Term    = end <ENTER>

yes
```

The following loop pops all terms from the stack and outputs them:

```
[user] ?- repeat,<ENTER>
            pop_global(stack,Term),<ENTER>
            write(Term),nl,<ENTER>
```

```
          not current_global(stack), !.<ENTER>
end
line2
line1

Term    = line1 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

!/0, fail/0, true/0, for/3

Restore default values for current streams

reset_streams

The predicate `reset_streams/0` closes all opened input/output streams, except the default I/O streams that were opened at system startup.

In other words, all assignments for the current input, output, and error output streams are reset to their default values at system startup.

current input/output stream	Assignment to default stream
`current_input`	`user_input`
`current_output`	`user_output`
`current_error`	`user_error`

Hints

If the default streams were redefined with `stream__copy/2`, the changes cannot be undone with `reset_streams/0`.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

set_input/1, set_output/1, set_error/1, current_input/1, current_output/1, current_error/1, stream_copy/2

Remove a clause from database

retract(+*Clause* **) [@** +*Module* **]**

retract(+*Head*, **?***Body* **) [@** +*Module* **]**

The predicate `retract/1` searches the database in the calling module or in the specified *Module* for clauses which can be unified with *Clause*. If such a clause is found, unification of *Clause* is performed, i.e. any existing variables in *Clause* are unified with the corresponding arguments (or their subterms) in the clause found. The first clause found is then deleted from the database.

The predicate `retract/2` searches the database in the calling module or in the specified *Module* for clauses with a head which can be unified with *Head*. If such a clause is found, unification of *Head* is performed, i.e. any variables in *Head* are unified with the corresponding arguments from the head of the clause found, and *Body* is unified with the body of the clause found. The first clause found is then deleted from the database.

The predicates `retract/1/2` succeed by backtracking for all matching clauses which were found in the database at the time of the first call, even if they have since been modified (**logic update view**).

Arguments

Clause	Structure, atom
Head	Structure, atom
Body	Term

Exceptions

instantiation_error
: The argument *Head* or *Clause* must not be a variable, but a variable was specified.

type_error(callable)
: The argument *Head* or the head of *Clause* must have the syntactical structure of a clause head.

permission_error(modify,static_procedure)
: The argument *Head* or the head of *Clause* specifies an existing predicate which has not been declared as dynamic and therefore cannot be modified.

Hints

Facts in the database have the clause body `true`.

If all clauses of a predicate are removed with `retract/1/2`, the predicate will remain defined, but will always fail. The predicate `abolish/1` can be used to delete a predicate.

The predicate `retract/1/2` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

Standard

Only the predicate `retract/1` complies with the definition of the ISO standard for Prolog.

See also

abolish/1, asserta/1/2, assertz/1/2, clause/2, retract_with_names/2/3, modify_mode/3

Remove a clause from database

retract_with_names(+*Clause*, ?*VarList* **)** `[` **@** +*Module* `]`

retract_with_names(+*Head*, ?*Body*, ?*VarList*) `[` **@** +*Module* `]`

The predicates `retract_with_names/2/3` act in the same way as `retract/1/2`, except that any name information on variables is also unified with *VarList* in the form:

`[` *Name* `=` *Variable*`, ... ]`

If there is no such information, *VarList* is unified with the empty list.

The predicates `retract_with_names/2/3` succeed by backtracking for all matching clauses which were found in the database at the time of the first call, even if they have since been modified (**logic update view**).

Arguments

Clause	Structure, atom
Head	Structure, atom
Body	Term
VarList	List in the form [*Name* = *Variable*,...]
Name	Atom
Variable	Variable

Exceptions

instantiation_error
The argument *Head* or *Clause* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Head* or the head of *Clause* must have the syntactical structure of a clause head.

permission_error(modify,static_procedure)
The argument *Head* or the head of *Clause* specifies an existing predicate which has not been declared as dynamic and therefore cannot be modified.

type_error(list)
The argument *VarList* must be a variable or a list, but is a term of another type.

type_error(assignment)
An element of *VarList* is not an assignment with the format *Name* `=` *Variable*.

type_error(atom)
The argument *Name* must be a variable or an atom, but is a term of another type.

type_error(variable)
The argument *Variable* must be a variable, but is a term of another type.

Hints

If all clauses of a predicate are removed with `retract_with_names/2/3`, the predicate will remain defined, but will always fail. The predicate `abolish/1` can be used to delete a predicate.

The predicate `retract_with_names/2/3` is a metapredicate and manipulates the database in the calling module or in the specified *Module*.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

abolish/1, asserta_with_names/2/3, assertz_with_names/2/3, clause_with_names/2/3, retract/1/2, modify_mode/3

Reverse a list

reverse(@*List*, ?*ReversedList* **)**

The predicate `reverse/2` reverses *List* and unifies *ReversedList* with the reversed list.

Arguments

List	List
ReversedList	List

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(list)
The argument *ReversedList* must be a variable or a list, but is a term of another type.

Example

```
[user] ?- reverse([z,e,u,s],R).<ENTER>

R       = [s,u,e,z] <ENTER>

yes

[user] ?- reverse([2,atom,5.6,f(g),Var],L).<ENTER>

Var     = _70
L       = [_70,f(g),5.6,atom,2] <ENTER>

yes

[user] ?- reverse([],X).<ENTER>

X       = [] <ENTER>

yes
```

```
[user] ?- Palindrome = [s,a,i,p,p,u,a,k,a,u,p,p,i,a,s],<ENTER>
          reverse(Palindrome, Palindrome).<ENTER>

Palindrome      = [s,a,i,p,p,u,a,k,a,u,p,p,i,a,s] <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

append/3, member/2

Store current module in a file

save_module(+*Filename* **)** `[` **@** +*Module* `]`

The predicate `save_module/1` saves the calling module or the specified *Module* in the file named *Filename*.

If *Filename* does not contain the extension *.cmp*, the extension is appended and the resultant filename *Filename.cmp* is used.

Arguments

Filename	Atom

Exceptions

instantiation_error
: The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
: The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
: The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
: The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
: The indicated error was reported by the operating system when writing on the output stream *Filename*.

Hints

Files created with `save_module/1` can be loaded by calling IF/Prolog with the parameter `-l` *Filename* or by using `load/1` during a Prolog session.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

load/1, unload/1, compile/1, save_system/1

Save system status for creating applications

save_runtime(+*Filename* **)**

The predicate `save_runtime/1` saves all static information concerning the current system status in the file *Filename*.

If *Filename* does not contain the extension *.pst*, the extension is appended and the resultant filename *Filename.pst* is used.

This includes status information on the

- database
- global variables
- operators
- modules

In contrast to `save_system/1`, the saved status does not contain any data which may only be used by the development system (e.g. `break/0`, `debug/1`, `trace/1`).
The saved state can be subsequently restored by calling IF/Prolog with the parameter `-r` *Filename* or by using `load_system/1` during a Prolog session.

Arguments

Filename	Atom

Exceptions

instantiation_error
: The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
: The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
: The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
: The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
: The indicated error was reported by the operating system when writing on the output stream *Filename*.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

save_system/1, load_system/1, save_module/1

Save current system status

save_system(+*Filename* **)**

The predicate `save_system/1` saves all static information concerning the current system status in the file *Filename*.

If *Filename* does not contain the extension *.pst*, the extension is appended and the resultant filename *Filename.pst* is used.

This includes status information on the

- database
- global variables
- operators
- modules

The saved state can be subsequently restored by calling IF/Prolog with the parameter `-r` *Filename* or by using `load_system/1` during a Prolog session.

Arguments

Filename	Atom

Exceptions

instantiation_error
The argument *Filename* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Filename*.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

save_runtime/1, load_system/1

Set current input stream

see(@*Stream* **)**

The predicate `see/1` makes *Stream* the current input stream.

If the argument *Stream* is not an input/output stream and not a defined alias, it must be an atom. A file of this name is then opened for reading. If this succeeds, the name is simultaneously registered as an alias for the opened input stream and the stream becomes the current input stream.

Arguments

Stream	Stream (system-defined term), alias (atom) or filename (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

domain_error(file_name)
The value of the argument *Stream* is not a valid name for a file.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(open,source_sink)
The argument *Stream* specifies a file (or a different I/O medium) that cannot be opened.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

Hints

All input predicates that have no explicit stream argument refer to the current input stream. This stream can always be addressed via the built-in alias `current_input` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

seen/0, seeing/1, set_input/1, error_tell/1, tell/1, open/3/4, assign_alias/2

Query current input stream

seeing(?*Stream*)

The predicate `seeing/1` unifies *Stream* with the `current` input stream.

The current input stream may be a stream that was set with `see/1` or `set_input/1`.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

domain_error(stream_or_alias)
: The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

Hints

The predicate `seeing/1` can unify *Stream* with an alias or the filename that was possibly used with `see/1` or `open/4`.

All input predicates that have no explicit stream argument refer to the current input stream. This stream can always be addressed via the built-in alias `current_input` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

see/1, seen/0, set_input/1, current_input/1, error_telling/1, telling/1

Reset current input stream

seen

The predicate `seen/0` resets the default input stream `user_input` as the current input stream.

If the earlier current input stream was not a default stream, it is closed (see `close/1`).

Hints

All input predicates that have no explicit stream argument refer to the current input stream. This stream can always be addressed via the built-in alias `current_input` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

see/1, seeing/1, current_input/1, set_input/1, close/1/2

Synchronous input multiplexing

select(@*Streams*, +*Timeout*, -*ReadyReadStreams* **)**

select(@*Streams*, +*Timeout*, -*ReadyReadStreams*, -*ReadyWriteStreams* **)**

select(@*SocksOrStreams*, +*Timeout*, -*ReadyReadSOS*, -*ReadyWriteSOS*, -*Timeleft* **)**

The predicate `select/3` examines the streams in the list *Streams* for readable data. Each element of *Streams* must be a stream opened for reading. The predicate `select/4` examines the streams in the list *Streams* for readable or writable data. Each stream element of *SocksOrStreams* must be either a socket or a stream opened for reading or opened for writing. The predicate `select/5` examines the sockets or streams in the list *SocksOrStreams* for readable or writable data. Each stream element of *SocksOrStreams* must be a stream opened for reading or opened for writing. The argument *Timeout* specifies the interval to wait for the selection to complete. If *Timeout* is the atom `off`, the predicate blocks, i.e. waits infinitely until something is available (or is interrupted). If *Timeout* is zero, the predicate polls the streams and returns immediately. Otherwise, *Timeout* must be a positive floating point number specifying the interval (in seconds) to wait if no socket or stream returns data.

The argument *ReadyReadStreams* resp. *ReadyReadSOS* must be a variable. It is instantiated to the sublist of streams or sockets of *Streams* resp. *SocksOrStreams* with readable data.

The argument *ReadyWriteStreams* resp. *ReadyWriteSOS* must be a variable. It is instantiated to the sublist of streams or sockets of *Streams* resp. *SocksOrStreams* with writable data.

If there is a timeout, *ReadyReadSOS* resp. *ReadyWriteSOS* is instantiated to the empty list.

The argument *TimeLeft* must be a variable. It is instantiated to the remaining part of time left of the *Timeout* interval (in seconds) if a socket or stream returns earlier. In default case *TimeLeft* returns zero.

Arguments

Streams	List of streams
SocksOrStreams	List of sockets or streams
Timeout	Positive floating point number or atom `off`
ReadyReadStreams	List of streams
ReadyWriteStreams	List of streams
ReadyReadSOS	List of sockets or streams
ReadyWriteSOS	List of sockets or streams
TimeLeft	Variable: positive integer or zero

Exceptions

instantiation_error
The argument *Streams* resp. *SocksOrStreams* or an element thereof must not be a variable, but a variable was specified.

type_error(list)
The argument *Streams* resp. *SocksOrStreams* must be a list, but is a term of another type.

domain_error(stream_or_alias)
The argument *Streams* must be a list and each element of it must be an input/-output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

instantiation_error
The argument *Timeout* must not be a variable, but a variable was specified.

type_error(float)
The argument *Timeout* must be a number, which can be converted implicitly into a floating-point number, but is a term of another type.

domain_error(not_less_than_zero)
The argument *Timeout* must be a positive number, but is negative.

type_error(variable)
The argument *ReadyReadStreams* or *ReadyWriteStreams* must be a variable, but is a term of another type.

type_error(variable)
The argument *ReadyReadSOS* or *ReadyWriteSOS* must be a variable, but is a term of another type.

type_error(variable)
The argument *Timeleft* must be a variable, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1B The predicate `select/5` is new.

See also

alarm/1, socket/3, current_socket/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5

Set array element

set_array(@*Array*, @*Index*, @*Term*)

The predicate `set_array/3` sets the element of *Array*, whose index is *Index*, to *Term*. The old value of the element is overwritten by the new value. The element will be reset to the old value on backtracking.

Arguments

Array	Array (system-defined term)
Index	Positive integer
Term	Term

Exceptions

instantiation_error
The argument *Array* or *Index* must not be a variable, but a variable was specified.

type_error(array)
The argument *Array* must be an array, but is a term of another type.

type_error(integer)
The argument *Index* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(array_size)
The argument *Index* must be a positive integer less than or equal to the number of elements in the array, but is not positive or larger.

Furthermore, if an arithmetic expression is specified for the argument *Index*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- create_array(Array, 3),
        set_array(Array, 1, 100),
        for(1, X, 2),
                get_array(Array, 1, Value1),
                write(first : Value1), nl,
                set_array(Array, 1, 200),
                get_array(Array, 1, Value2),
                write(second : Value2), nl,
        X = 2.
```

```
first : 100
second : 200
first : 100
second : 200

Array   = '$array'(200,0,0)
X       = 2
Value1  = 100
Value2  = 200 <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `set_array/3` is new.

See also

arg/3, array/1/2, create_array/2, get_array/3

Set current module

set_default_module(+*Module* **)**

The predicate `set_default_module/1` sets the *current* module to *Module*.

All goals entered in the input loop of IF/Prolog (see `break/0`) are executed in the context of the current module.

Arguments

Module	Atom, name of a module

Exceptions

instantiation_error
The argument *Module* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

existence_error(module)
The argument *Module* is not the name of an existing module.

existence_error(module_body)
The argument *Module* specifies a module for which no body exists.

permission_error(access,system_module)
The predicate `set_default_module/1` must not be executed for a system module.

Hints

When IF/Prolog starts, the built-in module `user` is the current module.

In the input loop of IF/Prolog (`break/0`), the current module is always shown in the prompt.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

module/1, current_module/1, current_default_module/1

Set current error output stream

set_error(@*Stream*)

The predicate `set_error/1` makes *Stream* the current error output stream. This stream is used by IF/Prolog for the output of error messages and warnings and is accessible under the alias `current_error`.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

Example

```
[user] ?- X is 4 / 0.<ENTER>

*** E X C E P T I O N: evaluation_error(zero_divisor)

no

[user] ?- open(null(null), write, BlackHole),<ENTER>
          set_error(BlackHole).

BlackHole       = '$stream'(5) <ENTER>

yes

[user] ?- X is 4 / 0.<ENTER>
```

```
no
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

set_input/1, set_output/1, current_input/1, current_output/1, current_error/1

Set value of a global variable

set_global(+*Name*, @*Value* **)** [@ +*Module*]

The predicate `set_global/2` sets the global variable *Name* in the calling module or in the specified *Module* to the given *Value*.

If the global variable *Name* does not exist, it is created.

If the global variable *Name* has a value stack, all previous values are removed from the stack, thus making *Value* the only value in the stack.

Arguments

Name	Atom
Value	Term

Exceptions

instantiation_error
The argument *Name* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

Hints

Global variables are always assigned to a module; they can be neither exported nor imported. In operations with global variables, therefore, only those global variables defined in the calling module or in the specified *Module* are taken into account.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_global/1, get_global/2, pop_global/2, push_global/2, unset_global/1

Set current input stream

set_input(@*Stream*)

The predicate `set_input/1` makes *Stream* the current input stream. This stream is used by all input predicates which are specified without an explicit stream argument and is accessible under the alias `current_input`.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

set_error/1, set_output/1, current_input/1, current_output/1, current_error/1

Set memory management parameters

set_memory_management(+*Area*, +*Parameter*, +*Value*)

The predicate `set_memory_management/3` sets the value of the parameter *Parameter* relating to memory management for the memory area *Area*. IF/Prolog manages the memory areas `database`, `global`, `constraints`, `local`, `trail` and `extension`. Each possesses the parameters `minimal_size`, `maximal_size`, `minimal_gap`, `maximal_gap`, `default_gap` and `trace`. The values of all but the last parameter are positive integers. The parameter `trace` has the value `on` or `off`.

The parameters have the following meaning:

`minimal_size`
: Minimum size of the memory area in bytes.

`maximal_size`
: Maximum size of the memory area in bytes; 0 if unlimited.

`default_gap`
: The value specified is interpreted as a percentage and is automatically limited to the range 0 - 1000. After garbage collection, the amount of free memory in the area must be at least `default_gap` percent of the memory used; otherwise the size of the memory area is automatically increased. The size of the free memory is further restricted by the parameters `minimal_gap` and `maximal_gap`.

  If `maximal_gap` is equal to 0 the size of free memory equals

```
min(
        max( memory_used / 100 * default_gap, minimal_gap),
        maximal_gap
) ,
```

  otherwise

```
max( memory_used / 100 * default_gap, minimal_gap) .
```

`minimal_gap`
: After garbage collection, at least `minimal_gap` bytes of free memory must be available in the area; otherwise the size of the data area is automatically increased.

`maximal_gap`
: After garbage collection, a maximum of `maximal_gap` bytes of free memory must be available; otherwise the size of the data area is automatically decreased. If this parameter is equal to 0, its value is not involved in calculating the free memory.

`trace`
: Activate/deactivate tracing of the garbage collection on the output stream `current_error`.

Arguments

Area	Atom: database | global | constraints | local | trail | extension
Parameter	Atom: minimal_size | maximal_size | minimal_gap | maximal_gap | default_gap | trace
Value	Integer or integer expression or atom (*Value* = on | off)

Exceptions

instantiation_error
The argument *Area*, *Parameter* or *Value* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Area*, *Parameter* or *Value* must be an atom, but is a term of another type.

domain_error(memory_area)
The argument *Area* must specify a memory area of the Prolog system. The specified atom is not a valid identifier.

domain_error(memory_parameter)
The value of the argument *Parameter* must be a configuration parameter of the Prolog memory management. The specified atom is not a valid identifier.

domain_error(on_or_off)
The argument *Value* must be one of the atoms `on` or `off`, but is another atom.

type_error(integer)
The argument *Value* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Value* must be a positive integer, but is negative.

Furthermore, if an arithmetic expression is specified for the argument *Value*, all the exceptions for `is/2` can occur.

Hints

The values actually set by IF/Prolog can deviate from the values specified as arguments due to alignment performed to improve memory usage.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_memory_management/3, garbage_collection/1

Set current output stream

set_output(@*Stream*)

The predicate `set_output/1` makes *Stream* the current output stream. This stream is used by all output predicates which are specified without an explicit stream argument and is accessible under the alias `current_output`.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

set_error/1, set_input/1, current_input/1, current_output/1, current_error/1

Set value of a Prolog flag

set_prolog_flag(+*Flag*, +*Value* **)**

The predicate `set_prolog_flag/2` sets the indicated Prolog *Flag* to *Value*.

Prolog flags are used to make global settings for IF/Prolog.

The following table contains the available Prolog flags and their permissible values (default values printed in **boldface**).

The following Prolog flags may be queried and modified:

Flag	Value	Description
`consult`	**notify**	A message is output when a Prolog text is loaded.
	`nonotify`	No message is output when a Prolog text is loaded.
`search_path`	*Paths*	If input files are not specified using a complete path name, they are searched for in the directories specified in the *Paths* list. The default value of this Prolog flag is the value of the system parameter `-sp`, if specified, otherwise it is the value of the environment variable `PROPATH` or it is empty.
`read_error`	**error**	Syntax errors found by analysis with `read_term/2/3`, etc. raise an exception.
	`fail`	Syntax errors found by analysis with `read_term/2/3`, etc. are reported directly by the parser in a message and the predicate fails.
	`quiet`	Syntax errors found by analysis with `read_term/2/3`, etc. are not reported and the predicate fails.
`report_read_error`		
	on	Syntax errors found by analysis with `parse_atom/6` are reported directly by the parser in a message.
	`off`	Syntax errors found by analysis with `parse_atom/6` are not reported by the parser but can be queried with the `read_error/2/3` predicates.
`warnings`	**on**	For `consult/1`, etc. warnings are output for possible errors.
	`off`	For `consult/1`, etc. no warnings are output.

`unknown`	**error**	If a predicate is to be activated which is not defined, an exception is raised. This does not apply to predicates for which the exception mode has been set to `on` (see `predicate_mode/3`).
	`fail`	If a predicate is to be activated which is not defined, backtracking is initiated, i.e. the predicate is evaluated with `fail`.
	`warning`	If a predicate is to be activated which is not defined, a warning is output and backtracking is initiated, i.e. this predicate is evaluated with `fail`. The warning is not output for predicates for which the exception mode has been set to `on` (see `predicate_mode/3`).
`debug`	`on`	For `consult/1` and `reconsult/1`, *all* predicates are implicitly declared as dynamic (see `dynamic/1`) to support program testing.
	off	For `consult/1` and `reconsult/1`, all the predicates which were not explicitly declared with `dynamic/1` are declared as static. These cannot then be modified, tested or displayed.
`prompt`	`on`	Output of the prompt in the Prolog input loop and interaction for backtracking for goals which have been successfully executed in the Prolog input loop and for which variables have been instantiated *always* take place, i.e. even if the standard input and the standard output are not terminals.
	off	Output of the prompt in the Prolog input loop and interaction for backtracking for goals which have been successfully executed in the Prolog input loop and for which variables have been instantiated take place *only* if the standard input and the standard output are terminals.
`char_conversion`	**on**	The character conversions defined with the predicate `char_conversion/2` are performed when terms are read.
	`off`	No character conversion is performed when terms are read.

`backtrace`	**10**	The maximal length of the **backtrace**-list. The list is created when an exception has been raised and contains the calling sequence of goals that has led to the exception. The elements of the list are predicate indicators (*Functor*/*Arity*). Goals that have been removed due to tail-recursion optimization (normally the last subgoal in a clause body) are not included in the list.
`double_quotes`	**codes**	Characters inside double quotes (") are to be interpreted as a list of character codes when reading a term (e.g. with `read/1/2`).
	`chars`	Characters inside double quotes (") are to be interpreted as a list of characters when reading a term (e.g. with `read/1/2`).
	`atom`	Characters inside double quotes (") are to be interpreted as an atom when reading a term (e.g. with `read/1/2`).
`extended_syntax`	`on`	Cyclic terms can be entered using the notation *Variable* `::` *Term*. When reading such a cyclic term, *Variable* is unified with *Term*. The above notation is also used by output of cyclic terms (see `=/2`).
	off	Cyclic terms can be entered in the above notation, but *Variable* is **not** unified with *Term*.
`nested_comments`	**off**	Nested comments are not allowed in a Prolog text. Begin of a comment inside a comment is ignored, which normally leads to a syntax error.
	`on`	Nested comments are accepted in a Prolog text.
`write_depth`	**10**	Maximal depth of terms output in the Prolog toplevel (see `break/0`). This applies to variable instantiations of a query and to exception messages. The predicate `write_formatted/2/3` takes this value in account, if a term is to be output with the format `%w`.

Arguments

Flag	Atom
Value	Atom, integer or integer expression

Exceptions

instantiation_error
The argument *Flag* or *Value* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Flag* must be an atom, but is a term of another type.

domain_error(prolog_flag)
The value of the argument *Flag* is not a valid Prolog flag.

domain_error(flag_value)
The value of the argument *Value* is none of the values valid for a Prolog flag.

permission_error(modify,flag)
The argument *Flag* is the name of a Prolog flag, the value of which may not be modified.

Hints

The default settings for the Prolog flags `consult`, `debug`, `prompt`, `search_path` and `warnings` can be modified by system parameters (see `system_parameters/1`) when IF/Prolog is started.

The values of the Prolog flags `notty`, `iso` and `signal` can be set using the appropriate system parameters **only** when IF/Prolog is started.

Example

```
[user] ?- set_prolog_flag(consult,notify).<ENTER>

yes

[user] ?- [user].<ENTER>
<EOF>
*** consult 'user': loaded in 0.00 sec.

yes

[user] ?- set_prolog_flag(consult,nonotify).<ENTER>

yes

[user] ?- [user].<ENTER>
<EOF>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The Prolog flags `backtrace`, `consult`, `double_quotes`, `extended_syntax`, `float_max`, `float_min`, `iso`, `nested_comments`, `notty`, `prompt`, `read_error`, `report_read_error`, `search_path`, `signal`, `warnings` and `write_depth` are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_prolog_flag/2, system_parameters/1

Set value of a Prolog flag

:- set_prolog_flag(+*Flag*, +*Value* **)**

The directive `set_prolog_flag/2` is analogous to the predicate `set_prolog_flag/2`. As as directive, it is executed by `consult/1`, `reconsult/1` or `compile/1` when a Prolog source text is read and takes effect immediately.

Arguments

Flag	Atom
Value	Atom, integer or integer expression

Exceptions

Errors in directives are detected with `consult/1`, `reconsult/1` or `compile/1`. No exceptions are generated. Instead, only error messages are output on the current error output stream.

This directive can produce the following error messages:

instantiation_error
The argument *Flag* or *Value* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Flag* must be an atom, but is a term of another type.

domain_error(prolog_flag)
The value of the argument *Flag* is not a valid Prolog flag.

domain_error(flag_value)
The value of the argument *Value* is none of the values valid for a Prolog flag.

permission_error(modify,flag)
The argument *Flag* is the name of a Prolog flag, the value of which may not be modified.

Standard

The directive `set_prolog_flag/2` complies with the definition of the ISO standard for Prolog.

The Prolog flags `backtrace`, `consult`, `double_quotes`, `extended_syntax`, `float_max`, `float_min`, `iso`, `nested_comments`, `notty`, `prompt`, `read_error`, `report_read_error`, `search_path`, `signal`, `warnings` and `write_depth` are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_prolog_flag/2

Set socket option

set_socket_option(@*Socket*, +*Option*, +*Value*)

The predicate `set_socket_option/3` sets an option for a communications socket.

The argument *Socket* must be a term representing a socket.

The set of supported options and corresponding values is system-dependent. The following table contains the allowed values for *Option*. The predicate **set_socket_option/3** succeeds, even if the given option *Option* is not supported for the underlying operating system.

The predicate **get_socket_option/3** can be used to test the current setting of options.

Option	Explanation	Type of *Value*
broadcast	Transmit broadcast messages	on | off
close_on_exec	Control take-over of current sockets	on | off
debug	Record debugging information	on | off
dontroute	Routing bypass for outgoing messages	on | off
keepalive	Keep connections alive	on | off
nonblocking	Make socket asynchroneous/synchroneous	on | off
oobinline	Reception of out-of-band data	on | off
linger	Linger on close if data present	integer
rcvbuf	Size of input buffer	integer
reuseaddr	Local address reuse	on | off
reuseport	Local port reuse	on | off
selectread	Designate socket for reading	on | off
selectwrite	Designate socket for writing	on | off
sndbuf	Size of output buffer	integer

Arguments

Socket	Socket (system-defined term)
Option	Atom
Value	Integer or atom: on | off

Exceptions

instantiation_error
The argument *Socket* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error(atom)
The argument *Option* must be an atom, but is a term of another type.

domain_error(socket_option)
The argument *Option* is not in the prescribed range of values.

type_error(integer)
The argument *Value* must be an integer, but is a term of another type.

type_error(atom)
The argument *Value* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Value* must be one of the atoms `on` or `off`, but is another atom.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_socket/3, get_socket_option/3, net_service/3, select/3, socket/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Position within an input/output stream

set_stream_position(@*Stream*, @*Position*)

The predicate `set_stream_position/2` sets the current position of the specified I/O stream *Stream* to the given *Position*, provided the device driver associated with *Stream* allows such positioning.

Positioning is possible only if the stream was opened with the option `reposition` (see `open/3/4`).

The *Position* is specified as a device-specific term and should hence only be a value that was previously determined with `stream_property/2`.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Position	Term, device-specific

Exceptions

instantiation_error
The argument *Stream* or *Position* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

domain_error(stream_position)
The argument *Position* is not a position specification permitted for setting the I/O stream as supplied by `stream_property/2`.

permission_error(reposition,stream)
The argument *Stream* is an I/O stream which does not permit positioning.

Hints

Positioning is realized by corresponding stream control functions of the device drivers (see `stream_control/2`).

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

open/3/4, stream_property/2

Find selected solutions of a goal

setof(?*Term*, +*Goal*, ?*TermList* **)** `[` **@** +*Module* `]`

The predicate `setof/3` unifies *TermList* with a sorted and duplicate-free list (see `sort/2`) of all instantiations of *Term*, with which *Goal* succeeds.

Term and *Goal* should share common variables.

Backtracking can take place by the variables which occur in *Goal* and *not* in *Term* (known as *free* variables). If `setof/3` succeeds, the variables occurring both in *Term* and in *Goal* remain uninstantiated.

If *Goal* is specified as

Term1 `^` *Term2* `^` ... `^` *SubGoal*

SubGoal is executed and the variables in *Term1*, *Term2* are no longer considered free, even if they occur only in *Goal* and not in *Term*. Backtracking, as described above, does not therefore take place by these variables.

Arguments

Term	Term
Goal	Goal
TermList	List
*Term*i	Term
SubGoal	Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

type_error(list)
The argument *TermList* must be a variable or a list, but is a term of another type.

Hints

The atom `'^'` is defined as an infix operator.

The predicate `setof/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- setof(X,member(X,[b,a,c,b,a]),L).<ENTER>

X       = _68
L       = [a,b,c] <ENTER>

yes

[user] ?- setof(X,(X=1;X=2;X=1),S).<ENTER>

X       = _152
S       = [1,2] <ENTER>

yes

[user] ?- setof(X,Y^((X=1,Y=1);(X=2,Y=2);(X=1,Y=3)),S).<ENTER>

X       = _152
S       = [1,2] <ENTER>

yes

[user] ?- setof(X,current_op(V,fy,X),L).<ENTER>

X       = _68
```

```
V       = 200
L       = [+,-,\] ;<ENTER>

X       = _68
V       = 900
L       = [\+,not] ;<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

call/1, findall/3, bagof/3, @/2, :/2, set_prolog_flag/2

Control signal processing

signal_control(+*Signal*, +*Command* **)**

The predicate `signal_control/2` enables the user to control signal processing by specifying the following *Command*s:

`default` Activates the **default** signal-handling mechanism of the operating system for *Signal*. In most cases, this involves aborting the process when the signal is received. The predicate `signal_control/2` is always successful with this *Command*.

`ignore` The signal-handling mechanism of the operation system for *Signal* is deactivated. The predicate `signal_control/2` is always successful with this *Command*.

`off` Activates the **passive** signal-handling mechanism of IF/Prolog for *Signal*. This means that arrivals of the specified *Signal* are registered and counted, but no action is taken. The predicate `signal_control/2` is always successful with this *Command*.

`on` Activates the **active** signal-handling mechanism of IF/Prolog for *Signal*. In other words, if *Signal* is received or was registered earlier by **passive** signal-handling, a user-defined signal handling routine is activated (if defined with `signal_handler/3` or `context/2`) or an exception of the form

```
error(system_error, [error=interrupt, signal=Signal,
                     goal=Goal])
```

is created. If a predicate implemented in C is interrupted, the entry `interrupted=`*Predicate* is added to the above-mentioned list. The predicate `signal_control/2` is always successful with this *Command*.

`clear` Resets the counter for incoming signals of the type *Signal* to 0 as if the signal had never arrived, provided passive signal handling (mode `off`) has been activated for *Signal*. The predicate `signal_control/2` is always successful with this *Command*.

`received` The predicate `signal_control/2` succeeds, if passive signal handling (mode `off`) has been activated for *Signal* and the arrival of *Signal* was registered, i.e. the counter for this *Signal* has a value greater than 0. The counter for the arrival of *Signal* is decremented by 1. Otherwise, the predicate fails.

`received`(*?Name*)
The predicate `signal_control/2` succeeds, if passive signal handling (mode `off`) has been activated for *Signal* and the arrival of *Signal* was registered,

i.e. the counter for this *Signal* has a value greater than 0. The counter for the arrival of *Signal* is decremented by 1. Otherwise, the predicate fails.

The argument *Name* is then unified with the name of the signal received. This is useful if `all` is specified for *Signal* since *Name* is then unified with the actual signal name.

Arguments

Signal	Atom: all | abort | alarm | interrupt | pipe | quit | termination | user_1 | user_2
Command	Atom or structure: on | off | default | ignore | clear | received | received(*Name*)

Exceptions

instantiation_error
The argument *Signal* or *Command* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Signal* must be an atom, but is a term of another type.

domain_error(signal_name)
The argument *Signal* must be one of the built-in signal names, but is another atom.

type_error(atom_or_compound)
The argument *Command* must be an atom or a structure, but is a term of another type.

domain_error(signal_control)
The value of the argument *Command* is none of the values permitted for a signal control command.

Hints

The pseudo-signal `all` can be used to execute *Command* for all signals. If `clear` or `received` is specified, this applies to all signals handled by the **passive** signal-handling mechanism of IF/Prolog at the time of the call.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

signal_handler/3, current_signal/2, context/2

Define local signal handling

signal_handler(+*Goal*, ?*Signal*, +*Handler* **)** [@ +*Module*]

The predicate `signal_handler/3` executes *Goal* and enables incoming signals to be handled during the goal processing.

If *Goal* succeeds, `signal_handler/3` is exited with success. By backtracking, *Goal* can supply several results.
If *Goal* does not (or does no longer) succeed, `signal_handler/3` also fails (i.e. backtracking is initiated).

If a signal is received while *Goal* is being executed, and the name of the signal received can be unified with *Signal*, execution of *Goal* is interrupted, and the specified *Handler* is called. After the *Handler* is successfully processed, the interrupted attempt to execute *Goal* is resumed.

If *Handler* fails, the exception

`error(system_error, [error=interrupt, signal=`*S*`, goal=`*G*`])`

is activated implicitly for the incoming signal *S* and the interrupted goal *G*. If a predicate implemented in C is interrupted, the entry `interrupted=`*Predicate* is added to the above-mentioned list.

Arguments

Goal	Goal
Signal	Atom: abort | alarm | interrupt | pipe | quit | termination | user_1 | user_2
Handler	Goal

Exceptions

instantiation_error
The argument *Goal* or *Handler* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Signal* must be a variable or an atom, but is a term of another type.

instantiation_error
The argument *Goal* or *Handler* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* or *Handler* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal* or *Handler*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or *Handler* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or *Handler* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

Exception handling of this nature is only permitted for signals for which **active** signal handling has been activated (see `signal_control/2`).

Before the *Handler* is activated, signal handling for the incoming signal is set to `off` (**passive** handling). In other words, signal handling must be reactivated from within the *Handler* if required.

The predicate `signal_handler/3` is mapped to `context/2`.

The predicate `signal_handler/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

call/1, current_signal/2, signal_control/2, context/2, set_prolog_flag/2, unix_kill/2

Skip an input line

skip_line

skip_line(@*Stream* **)**

The predicate `skip_line/0` reads characters from the *current* input stream (`current_input`) until and including the next end-of-line character or the end of the stream.

The predicate `skip_line/1` has the same effect, but on the specified input stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(input,stream)
The argument *Stream* is not a stream which has been opened for input (open mode `read`).

permission_error(past_end_of_stream)
The `open` option `eof_action(error)` was set for the input stream *Stream* and the end of the stream has already been reached.

system_error(read)
system_error(interrupted)
The error indicated was reported by the operating system when reading the input stream *Stream*.

Example

The following predicates could be used to skip comments:

```
[user] ?- read(T), skip_line, read(N).<ENTER>
gabba. Comment<ENTER>
next_line.<ENTER>

T       = gabba
N       = next_line <ENTER>

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

get_char/1/2, get_until/3/4

Create communication socket

socket(+*Domain*, +*Type*, -*Socket* **)**

The predicate `socket/3` creates an endpoint for communication.

The argument *Domain* specifies the communications domain within which communication will take place. If *Domain* is `unix`, communication uses UNIX protocols and is possible only between processes running on the same machine. If *Domain* is `internet`, communication uses DARPA Internet protocols and is also possible between processes running on different machines. Prolog allows up to 64 sockets existing simultaneously.

The argument *Type* specifies the semantics of the communication. If *Type* is `datagram`, communication is based on connectionless, unreliable messages of a fixed (typically small) maximum length. If *Type* is `stream`, communication is based on sequenced, reliable, two-way connection based byte streams. A `stream` socket must be in a `connected` state before any data may be sent or received on it (see `socket_accept/3` and `socket_connect/2`).

The argument *Socket* must be an uninstantiated variable. It will be instantiated to a system-generated term.

Arguments

Domain	Atom: internet | unix
Type	Atom: datagram | stream
Socket	Socket (system-defined term)

Exceptions

instantiation_error
The argument *Domain* or *Type* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Domain* or *Type* must be an atom, but is a term of another type.

domain_error(socket_domain)
The argument *Domain* does not specify a valid domain for a socket.

domain_error(socket_type)
The argument *Type* does not specify a valid type for a socket.

type_error(variable)
The argument *Socket* must be a variable, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

net_service/3, current_socket/3, socket_accept/3, socket_bind/2, socket_close/1, socket_connect/2, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Accept a connection

socket_accept(@*Socket*, ?*Addr*, -*NewSocket* **)**

The predicate `socket_accept/3` accepts a connection from a client.

The argument *Socket* specifies a `stream` socket opened by `socket/3`, which has been bound to an address by `socket_bind/2` and is listening to connections by `socket_listen/1/2`. `socket_accept/3` extracts the first connection on the queue of pending connections and creates a new socket with the same properties as *Socket*. The argument *NewSocket* is instantiated to the identification of the new socket.

The argument *Addr* is unified with the address of the connecting entity. The address for `internet` domain socket has the form ?*Host* : ?*Port*, where *Host* is the name of the receiving machine and *Port* is the port number. The address for `unix` domain socket is an atom *Filename*.

Arguments

Socket	Socket (system-defined term)
Addr	Structure: *Host* : *Port* or atom: *Filename*
Host	Atom
Port	Integer
Filename	Atom
NewSocket	Socket (system-defined term)

Exceptions

instantiation_error
The argument *Socket* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

domain_error(stream_socket)
The argument *Socket* must be a `stream` socket, but is a socket of another type.

domain_error(bound_socket)
The argument *Socket* does not specify a socket which has been bound to an address.

type_error('host:port')
The argument *Addr* must be a variable or a structure of the form *Host* : *Port* but is a term of another type.

type_error(atom)
The argument *Host* must be a variable or an atom, but is a term of another type.

type_error(integer)
The argument *Port* must be a variable or an integer, but is a term of another type.

type_error(atom)
The argument *Filename* must be a variable or an atom, but is a term of another type.

type_error(variable)
The argument *NewSocket* must be a variable, but is a term of another type.

system_error(too_many_sockets)
The system-imposed limit for the number of sockets has been exceeded.

Hints

This predicate is used only in a server process. The client process uses the predicate `socket_connect/2`.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

socket/3, current_socket/3, alarm/1, host_addr/2, socket_bind/2, socket_close/1, socket_connect/2, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Bind a name to a socket

socket_bind(@*Socket*, ?*Addr*)

The predicate `socket_bind/2` assigns a name to the socket *Socket*.

The argument *Addr* specifies the name to which the socket is to be assigned. The name for `internet` domain socket must have the form ?*Port*, where *Port* is the port number. If *Port* is uninstantiated, the system chooses a port and *Port* is unified with its number. The name for `unix` domain socket must be an atom `+`*Filename*. The file thus created will be deleted when the socket is closed by `socket_shutdown/1/2`.

Arguments

Socket	Socket (system-defined term)
Addr	Integer: *Port* or atom: *Filename*
Host	Atom
Port	Integer
Filename	Atom

Exceptions

instantiation_error
The argument *Socket* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error(integer)
The argument *Port* must be a variable or an integer, but is a term of another type.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

permission_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened.

Hints

This predicate is used normally in a server process. The client process uses the predicate `socket_connect/2`.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

socket/3, current_socket/3, current_host/1, host_addr/2, net_service/3, socket_accept/3, socket_close/1, socket_connect/2, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Close a socket

socket_close(@*Socket* **)**

The predicate `socket_close/1` causes to close, shut down and dismiss completely the socket *Socket*.

Arguments

Socket	Socket (system-defined term)

Exceptions

instantiation_error
The argument *Socket* or *How* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1B The predicates `socket_close/1` are new.

See also

socket/3, current_socket/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Connect a socket

socket_connect(@*Socket*, @*Addr*)

The predicate `socket_connect/2` connects the socket *Socket*.

The argument *Addr* specifies the address to which the socket is to be connected. The address for `internet` domain socket must have the form `+`*Host* : `+`*Port*, where *Host* is either the atom representing the hostname, or a single-quoted atom or a list of three integers, both representing the IP-Address, of the remote machine. *Port* is the port number of the remote machine. The address for `unix` domain socket must be an atom `+`*Filename*. The file must exist and be a socket, i.e. must have been created previously by the server process with `socket_bind/2`.

A `stream` socket may be connected only once. A `datagram` socket can be disconnected by specifying *Addr* = 0.

Arguments

Socket	Socket (system-defined term)
Addr	Structure: *Host* : *Port* or atom: *Filename*
Host	Atom or IntegerList
Port	Integer
Filename	Atom

Exceptions

instantiation_error
The argument *Socket* or *Addr* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error('host:port')
The argument *Addr* must be a structure of the form *Host* : *Port* but is a term of another type.

type_error(atom)
The argument *Host* must be an atom, but is a term of another type.

type_error(integer)
The argument *Port* must be an integer, but is a term of another type.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

existence_error(open,source_sink)
The argument *Filename* specifies a file (or a different I/O medium) that cannot be opened for reading.

domain_error(unix_socket_file)
The argument *Filename* must refer to a `unix` socket file, but the file is of another type.

Hints

This predicate is used only in a client process. The server process uses the predicates `socket_bind/2` and `socket_accept/3`.

If the connection cannot be established due to the exception

```
system_error('Connection refused')
```

then the application should close the socket using `socket_shutdown/1/2` and obtain a new socket with `socket/3` before attempting another connection.

If the socket is not designated as nonblocking (see `set_socket_option/3`) `socket_connect` may block further control.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_socket/3, socket/3, host_addr/2, net_service/3, socket_accept/3, socket_bind/2, socket_close/1, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2, set_socket_option/3

Listen for connections

socket_listen(@*Socket* **)**
socket_listen(@*Socket*, +*Quelen* **)**

The predicates `socket_listen/1/2` listen for connections on a socket.

The argument *Socket* specifies a `stream` socket opened by `socket/3`,

The argument *Quelen* specifies the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, the client may receive an error or the request may be ignored so that retries may succeed.

The predicate `socket_listen/1` acts like `socket_listen/2` with *Quelen* = 1.

Arguments

Socket	Socket (system-defined term)
Quelen	Integer

Exceptions

instantiation_error
The argument *Socket* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

domain_error(stream_socket)
The argument *Socket* must be a `stream` socket, but is a socket of another type.

type_error(integer)
The argument *Quelen* must be an integer, but is a term of another type.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

alarm/1, socket/3, current_socket/3, select/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Receive data from socket

socket_raw_receive(@*Socket*, @*Size*, ?*List*, ?*Length* **)**

socket_raw_receive(@*Socket*, @*Size*, ?*Addr*, ?*List*, ?*Length* **)**

The predicates `socket_raw_receive/4/5` receive data with buffer and length information from a socket. The predicate `socket_raw_receive/4` receives data from a connected socket (see `socket_connect/2` and `socket_accept/3`). The predicate `socket_raw_receive/5` receives a datagram from any address.

The argument *Socket* must be a term representing a socket.

The argument *Size* must be a positive integer expression, representing a buffer size for incoming data. Data are only received up to this buffer size. With a connected socket, the remaining data sent from the same source, if any, may be received by a subsequent `socket_raw_receive/4`. With a datagram socket, the remaining data are discarded immediately.

The argument *Addr* of `socket_raw_receive/5` is unified with the address from which data were received. The address for `internet` domain socket has the form ?*Host* : ?*Port*, where *Host* is the name of the receiving machine and *Port* is the port number. The address for `unix` domain socket is an atom *Filename*.

The argument *List* is unified with a list of character codes, including binary zero. If no data have been received yet, `socket_raw_receive/4/5` will block control until data arrive. Provisional inquiry whether data exist is possible by `select/4`.

The argument *Length* is unified with the integer length of the received data, up to the supplied buffer size.

If the receiving socket is designated as non-blocking (see `set_socket_option/3`) and nothing has been received, *List* will be unified with the empty list, *Length* will deliver zero, and processing continues.

Arguments

Socket	Socket (system-defined term)
Size	Integer expression
Addr	Structure: *Host* : *Port* or Atom: *Filename*
Host	Atom
Port	Integer
Filename	Atom
List	List of character codes
Length	Integer

Exceptions

instantiation_error
The argument *Socket* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error(integer)
The argument *Size* must be an integer, but is a term of another type.

type_error('host:port')
The argument *Addr* must be a variable or a structure of the form *Host* : *Port* but is a term of another type.

type_error(atom)
The argument *Host* must be a variable or an atom, but is a term of another type.

type_error(integer)
The argument *Port* must be a variable or an integer, but is a term of another type.

type_error(atom)
The argument *Filename* must be a variable or an atom, but is a term of another type.

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

type_error(integer)
The argument *Length* must be a variable or an integer, but is a term of another type.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1B The predicates `socket_raw_receive/4/5` are new.

See also

socket/3, current_socket/3, select/3, host_addr/2, net_service/3, set_socket_option/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_send/2/3, socket_shutdown/1/2

Receive data from socket

socket_receive(@*Socket*, ?*List* **)**

socket_receive(@*Socket*, ?*Addr*, ?*List* **)**

The predicates `socket_receive/2/3` receive data from a socket. The predicate `socket_receive/2` receives data from a connected socket (see `socket_connect/2` and `socket_accept/3`). The predicate `socket_receive/3` receives a datagram from any address.

The argument *Socket* must be a term representing a socket.

The argument *Addr* of `socket_receive/3` is unified with the address from which data was received. The address for `internet` domain socket has the form ?*Host* : ?*Port*, where *Host* is the name of the receiving machine and *Port* is the port number. The address for `unix` domain socket is an atom *Filename*.

The argument *List* is unified with a list of character codes. If nothing has been received, *List* will be unified with the empty list.

Arguments

Socket	Socket (system-defined term)
Addr	Structure: *Host* : *Port* or atom: *Filename*
Host	Atom
Port	Integer
Filename	Atom
List	List of integers

Exceptions

instantiation_error
The argument *Socket* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error('host:port')
The argument *Addr* must be a variable or a structure of the form *Host* : *Port* but is a term of another type.

type_error(atom)
The argument *Host* must be a variable or an atom, but is a term of another type.

type_error(integer)
The argument *Port* must be a variable or an integer, but is a term of another type.

type_error(atom)
The argument *Filename* must be a variable or an atom, but is a term of another type.

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

socket/3, current_socket/3, select/3, host_addr/2, net_service/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_raw_receive/4/5, socket_send/2/3, socket_shutdown/1/2

Send data to socket

socket_send(@*Socket*, @*List*)

socket_send(@*Socket*, @*Addr*, @*List*)

The predicates `socket_send/2/3` transmit data to a socket. The predicate `socket_send/2` sends data to a connected socket (see `socket_connect/2` and `socket_accept/3`). The predicate `socket_send/3` sends a datagram to the specified address.

The argument *Socket* must be a term representing a socket.

The argument *Addr* of `socket_send/3` is the address where the data is to be sent. The address for `internet` domain socket must have the form `+`*Host* : `+`*Port*, where *Host* is either the atom representing the hostname, or a single-quoted atom or a list of three integers, both representing the IP-Address, of the receiving machine and *Port* is the port number. The address for `unix` domain socket must be an atom `+`*Filename*.

The argument *List* must be a completely instantiated list of character codes.

Arguments

Socket	Socket (system-defined term)
Addr	Structure: *Host* : *Port* or atom: *Filename*
Host	Atom or IntegerList
Port	Integer
Filename	Atom
List	List of integers or integer expressions

Exceptions

instantiation_error
: The argument *Socket*, *Addr* or *List* or a subterm must not be a variable, but a variable was specified.

type_error(socket)
: The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
: The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error('host:port')
: The argument *Addr* must be a structure of the form *Host* : *Port* but is a term of another type.

type_error(atom)
: The argument *Host* must be an atom, but is a term of another type.

type_error(integer)
The argument *Port* must be an integer, but is a term of another type.

type_error(atom)
The argument *Filename* must be an atom, but is a term of another type.

domain_error(file_name)
The value of the argument *Filename* is not a valid name for a file.

domain_error(unix_socket_file)
The argument *Filename* must refer to a `unix` socket file, but the file is of another type.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(integer)
An element of *List* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

representation_error(character_code)
An element of *List* must be an integer representing a character code.

Furthermore, if an arithmetic expression is specified for an element in *List*, all the exceptions for `is/2` can occur.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

socket/3, current_socket/3, select/3, host_addr/2, net_service/3, socket_accept/3, socket_bind/2, socket_connect/2, socket_close/1, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_shutdown/1/2

Close a socket

socket_shutdown(@*Socket* **)**
socket_shutdown(@*Socket*, +*How* **)**

The predicate `socket_shutdown/2` causes all or part of a full-duplex connection on the socket *Socket* to be shut down. The socket may, however, continue existing in the operating system, especially if a partner still uses it. See **socket_close/1**.

The argument *How* specifies which part(s) of the connection is to be shut down:
If *How* is 0, then further receives will be disallowed.
If *How* is 1, then further sends will be disallowed.
If *How* is 2, then further sends and receives will be disallowed.

The predicate `socket_shutdown/1` acts as `socket_shutdown/2` with *How* = 2.

Arguments

Socket	Socket (system-defined term)
How	Integer 0, 1 or 2

Exceptions

instantiation_error
The argument *Socket* or *How* must not be a variable, but a variable was specified.

type_error(socket)
The argument *Socket* must be a socket supplied by `socket/3`, but is a term of another type.

domain_error(socket)
The argument *Socket* must be a currently open socket supplied by `socket/3`.

type_error(integer)
The argument *How* must be an integer, but is a term of another type.

domain_error(how)
The argument *How* is not in the prescribed range of values.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

socket/3, current_socket/3, socket_accept/3, socket_bind/2, socket_close/1, socket_connect/2, socket_listen/1/2, socket_receive/2/3, socket_raw_receive/4/5, socket_send/2/3

Sort a list

sort(@*List*, ?*SortedList* **)**

The predicate `sort/2` constructs a sorted list from the elements in *List* and unifies the new list with *SortedList*.

The collating sequence is determined by the order of terms in Prolog (see `@</2`) and is in ascending order.

If there are multiple occurrences of elements in *List*, each such element will appear only once in the sorted list.

Arguments

List	List
SortedList	List

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a list, but is a term of another type.

type_error(list)
The argument *SortedList* must be a variable or a list, but is a term of another type.

Example

Sorting occurs in the given order: variables, floating-point numbers, rational numbers, integers, atoms and structures:

```
[user] ?- sort([a(a,a),777,zzzz,X],L).<ENTER>

X       = _71
L       = [_71,777,zzzz,a(a,a)] <ENTER>

yes
```

Atoms are sorted in alphabetical order:

```
[user] ?- sort([c,b,a,d],L).<ENTER>

L       = [a,b,c,d] <ENTER>

yes
```

Numbers of equal type are sorted in arithmetic order:

```
[user] ?- sort([3,4,56,2],L).<ENTER>

L       = [2,3,4,56] <ENTER>

yes
```

As in the case of comparison predicates for terms, structures are sorted first by arity, then by the name of the functor, and finally by arguments from left to right:

```
[user] ?- sort([a(a,b),a(a,a),f(a)],L).<ENTER>

L       = [f(a),a(a,a),a(a,b)] <ENTER>

yes
```

Care must be taken with variables:

```
[user] ?- sort([Z,A],L), Z = zzz, A = aaa.<ENTER>

Z       = zzz
A       = aaa
L       = [zzz,aaa] <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

@</2, @>/2

System statistics

statistics

statistics(?*Type*, ?*ResultList*)

The predicate `statistics/0` writes system statistics on the output stream `current_output`.

The predicate `statistics/2` unifies *Type* and *ResultList* by backtracking with all the defined classes of system information and their associated values. *Type* is the system information class and *ResultList* is a list of the associated values with the format *Name* `=` *Value*. The following lists are generated, depending on *Type*:

`time`

`cputime =` *Value*
: CPU time used by the IF/Prolog process up to now.

`time =` *Value*
: System time in seconds.

`memory`

`system =` *Value*

`database =` *Value*

Stack `=` *Value*
: Size of the memory areas used by IF/Prolog.
`system` provides information on the total memory usage of IF/Prolog, `database` on the usage for the Prolog database and *Stack* the usage for the individual Prolog stacks. *Value* is unified with a list with the format `[`*Total*`,`*Used*`]`, where *Total* is the allocated memory and *Used* the used memory in bytes. *Stack* is unified with the name of a Prolog stack (`global`, `constraints`, `local`, `trail`, `extension`).

`garbage_collection`

`database =` *Value*

Stack `=` *Value*
: Information on garbage collection. `database` provides information on the garbage collection in the Prolog database, *Stack* provides information on the garbage collection in the individual Prolog stacks (`global`, `local`, `trail`, `extension`). *Value* is unified with a list with the format `[`*Number*`,`*Garbage*`,`*Time*`]` *Number* is the number of garbage collection calls for the associated memory area, *Garbage* is the amount of memory released (in bytes) and *Time* is the CPU time used for

the garbage collection (in seconds). No values are made available for the stack `constraints`, since the garbage collection of this stack is performed together with the garbage collection of the global stack. Accounting relating to the garbage collection for the constraint stack is included in the information for the global stack.

Arguments

Type	Atom: time | memory | garbage_collection
ResultList	List with the format [*Name* = *Value*, ...]
Name	Term
Value	Term

Exceptions

type_error(atom)
The argument *Type* must be a variable or an atom, but is a term of another type.

type_error(list)
The argument *ResultList* must be a variable or a list, but is a term of another type.

Example

Query system values:

```
[user] ?- statistics.<ENTER>

  memory statistic :     size      used    #gc collected   gc-time

          database :   700404    461648      1      1384      0.00
            global :    30720       412      0         0      0.00
       constraints :     8192         0
             local :    30720       396      0         0      0.00
         extension :     8192         0      0         0      0.00
             trail :     8192        20      0         0      0.00

             total :   786420    462476      1      1384      0.00

              date : Thu Jul 14 15:08:36 1994
           cputime : 1.63

yes
```

Query used time:

```
[user] ?- statistics(time,S).<ENTER>

S       = [cputime = 1.53333,time = 774191317] <ENTER>

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

set_memory_management/3, current_memory_management/3, is/2

Control an input/output stream

stream_control(@*Stream*, @*Command*)

The predicate `stream_control/2` enables device-dependent control of an I/O stream.

A device driver can provide a set of commands to execute specific operations on I/O streams. Many built-in predicates of IF/Prolog make use of these interfaces (e.g. to control the terminal).

New device drivers can be linked into IF/Prolog using the C driver interface. Built-in predicates using the control interface of drivers can operate on I/O streams for these new device drivers as well, provided the required control commands are supported by the drivers. The following control commands are implemented in the built-in device drivers of the Prolog system:

Device	Command	Meaning
`file`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`stream_size(`*N*`)`	Unifies *N* with the size of the file
	`isatty`	`stream_control/2` succeeds if *Stream* is a terminal
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file
	`clear_eos`	see `clear_eos/0/1`
	`clear_eol`	see `clear_eol/0/1`
	`clear_screen`	see `clear_screen/0/1`
	`tty_size(`*Z*`,`*S*`)`	see `tty_size/2/3`
	`move_cursor(`*Z*`,`*S*`)`	see `move_cursor/2/3`
	`seek(`*P*`)`	see `set_stream_position/2`
	`tell(`*P*`)`	see `stream_property/2`
`standard`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`stream_size(`*N*`)`	Unifies *N* with the size of the file
	`isatty`	`stream_control/2` succeeds if *Stream* is a terminal
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file
	`clear_eos`	see `clear_eos/0/1`
	`clear_eol`	see `clear_eol/0/1`
	`clear_screen`	see `clear_screen/0/1`
	`tty_size(`*Z*`,`*S*`)`	see `tty_size/2/3`
	`move_cursor(`*Z*`,`*S*`)`	see `move_cursor/2/3`
`pipe`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file

`socket`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file
`string`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`stream_size(`*N*`)`	Unifies *N* with the size of the string
	`string(`*Arg*`)`	Unifies *Arg* with the contents of the string
	`seek(`*P*`)`	see `set_stream_position/2`
	`tell(`*P*`)`	see `stream_property/2`
`null`	`stream_type(`*Arg*`)`	see `stream_type/2`
`-`	`get_last(`*Z*`,`*S*`)`	see `get_last/2/3`

Arguments

Stream	Stream (system-defined term) or alias (atom)
Command	Term, device-dependent

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

instantiation_error
The argument *Command* or a subterm must not be a variable, but a variable was specified.

type_error(atom_or_compound)
The argument *Command* must be an atom or a structure, but is a term of another type.

domain_error(stream_control)
The value of the argument *Command* is not any of the values permitted for a command for I/O stream control.

Hints

The command `get_last/2` is not supported by any of the built-in device drivers; if it is implemented in an additionally linked-in device driver, the predicate `get_last/2/3` can use this capability.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_stream_control/3, stream_device/2

Redefine streams

stream_copy(@*Stream1*, @*Stream2*)

The predicate `stream_copy/2` is primarily used to make a specified input/output stream to a standard stream during a IF/Prolog process. For example, if *Stream2* has the value `user_input`, `user_output` or `user_error`, *Stream1* will be redefined accordingly as the standard input, output or error output stream.

The predicate can also be used for any streams that have been opened for the same I/O mode (read or write), provided their device drivers support the required commands.

If *Stream2* is associated with an open file, it will be closed before *Stream1* is copied. Before that, `stream_control`(*Stream2*, `update`) is called, if the control command `update` is defined for this stream. This allows the device driver to perform cleanup tasks. After the operation, both streams will access the same file.

Arguments

Stream1	Stream (system-defined term) or alias (atom)
Stream2	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream1* or *Stream2* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream1* or *Stream2* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream1* or *Stream2* must be associated with an open input/output stream.

permission_error(duplicate,stream)
The argument *Stream1* or *Stream2* specifies an input/output stream that is not suitable for the required operation.

permission_error(input,stream)
The argument *Stream2* is not a stream which has been opened for input (open mode `read`). Both *Stream1* and *Stream2* must have been opened using the same mode.

permission_error(output,stream)
The argument *Stream2* is not a stream which has been opened for output (open

mode `write` or `append`), Both *Stream1* and *Stream2* must have been opened using the same mode.

Hints

This predicate is system dependent and may not be available on all operating systems.

The predicate `stream_copy/2` is based on the control commands for I/O streams `fileno` and `dup`. The standard device drivers `standard`, `file`, and `pipe` use the C function `dup(2)` for this purpose (see [10]).

Example

The file `example` contains the following lines:

```
append(A,B,[a,b,c,d]).
read(A).
-1.943.
a b.
write(test),nl.
```

If this file is defined as standard input after starting IF/Prolog, the goals from `example` will be subsequently processed as if they were entered at the terminal. When the end of the file is encountered, IF/Prolog is exited.

```
[user] ?- open(example,read,S),<ENTER>
          stream_copy(S,user_input).<ENTER>

S       = '$stream'(5)
yes


A       = []
B       = [a,b,c,d]
yes


A       = -1.943
yes

*** syntax_error: operator expected
>>> line 6, column 3

test

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

open/3/4, close/1/2, stream_control/2, current_stream_control/3

Determine device name of an input/output stream

stream_device(@*Stream*, ?*Device* **)**

The predicate `stream_device/2` unifies *Device* with the name of the device driver associated with *Stream*.

The following device drivers are normally built into IF/Prolog:

Device	Usage
`file`	Operating system files
`standard`	Standard I/O streams
`string`	Strings
`socket`	Sockets
`pipe`	Operating system pipes
`null`	Null device (in UNIX equivalent to `/dev/null`)

Some devices (e.g. `pipe`, `null`) may not be available on all operating systems.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Device	Atom

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

type_error(atom)
The argument *Device* must be a variable or an atom, but is a term of another type.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

open/3/4, stream_property/2, stream_type/2

Query information on input/output streams

stream_property(?*Stream*, ?*Info*)

The predicate `stream_property/2` unifies by backtracking its arguments with information on all opened input/output streams.

Most of this information corresponds to the `open/3/4` parameters. The following information can be obtained and unified with *Info*:

`file_name(Device(Name))`
: Corresponds to the first argument of `open/3/4`.

`input`
: The *Stream* was opened for reading (`read` mode with `open/3/4`).

`output`
: The *Stream* was opened for writing (`write` or `append` mode with `open/3/4`).

`type(`*Type*`)`
: *Type* is unified with the type of the stream corresponding to the option `type(`*Type*`)` with `open/4`.

`buffer(`*Buffering*`)`
: *Buffering* is unified with the type of buffering defined for *Stream* with `open/4`.

`reposition(`*Mode*`)`
: *Mode* is unified with `true` if repositioning of *Stream* is possible, otherwise with `false`.

`position(`*Position*`)`
: If repositioning is possible for *Stream*, *Position* is unified with the current position of the input/output stream; *Position* is a device-specific term that can be used with `set_stream_position/2`.

`alias(`*Alias*`)`
: *Alias* is unified with each named alias associated with *Stream*.

`eof_action(`*Action*`)`
: *Action* is unified with the type of handling defined for *Stream* with `open/4` on reaching the end of the stream.

`mode(`*Mode*`)`
: *Mode* is unified with the I/O-Mode which was specified when *Stream* was opened.

`end_of_stream(`*End*`)`
: If the stream position is `end_of_stream`, then *End* is unified with `at`; else if the stream position is `past_end_of_stream`, then *End* is unified with `past`; else *End* is unified with `not`.

The predicate `stream_property/2` succeeds by backtracking for all those I/O streams (and their associated properties) which were found at the time of the first call, even if they have since been closed or if new ones have been added (**logic update view**).

Arguments

Stream	Stream (system-defined term) or alias (atom)
Info	Structure or atom

Exceptions

domain_error(stream)
The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4`.

domain_error(stream_property)
The argument *Info* must be a variable, an atom or a structure denoting a stream property, but is a term of another type.

Hints

If the argument *Stream* is instantiated, it may be an alias. Since the ISO standard allows only a stream as the argument, this functionality is not available in the ISO compatibility mode. If *Stream* is a variable, `stream_property/2` will always instantiate it to an input/output stream and never to an alias.

Positioning is realized by corresponding stream control functions of the device drivers (see `stream_control/2`).

The sequence of solutions is undefined, if `stream_property/2` has several solutions for the current call.

Example

```
[user] ?- current_alias(user_input, S),
        stream_property(S,I).<ENTER>

S       = '$stream'(0)
I       = file_name(standard(user_input)) ;<ENTER>

I       = input ;<ENTER>

I       = mode(read) ;

I       = buffer(no) ;<ENTER>
```

```
I       = eof_action(reset) ;<ENTER>

I       = reposition(false) ;<ENTER>

I       = type(text) ;<ENTER>

I       = alias(user_input) ;<ENTER>

I       = alias(current_input) ;<ENTER>

I       = end_of_stream(not) ;<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The information `buffer(...)` is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.0B ISO conformance: the predicate `stream_property/2` rejects an alias as the first argument in ISO compatibility mode.

See also

open/3/4, close/1/2, current_alias/2, set_stream_position/2, stream_control/2, stream_device/2, stream_property/2

Query type of an input/output stream

stream_type(@*Stream*, ?*StreamType*)

The predicate `stream_type/2` unifies *StreamType* with the device-specific type of the specified input/output stream *Stream*.

The following types are possible for the built-in device drivers:

Device	Type	Description
`file`	`regular`	Regular file
	`block_special`	Special file
	`character_special`	Special file
	`directory`	Directory
	`fifo_special`	Named pipe
	`symbolic_link`	Symbolic link
	`unknown`	Unknown type
`standard`	`regular`	Regular file
	`block_special`	Special file
	`character_special`	Special file
	`directory`	Directory
	`fifo_special`	Named pipe
	`symbolic_link`	Symbolic link
	`unknown`	Unknown type
`pipe`	`pipe`	Pipe
`string`	`string`	String, atom
`socket`	`socket`	Socket
`null`	`null`	Null device (UNIX: `/dev/null`)

Arguments

Stream	Stream (system-defined term) or alias (atom)
StreamType	Atom

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

domain_error(stream_control)
The argument *Stream* is an input/output stream for which the required control command is not implemented in the device driver.

type_error(atom)
The argument *StreamType* must be a variable or an atom, but is a term of another type.

Hints

The predicate `stream_type/2` is based on the control command `stream_type/1` of the device drivers and can hence be used on I/O streams of new device drivers as well, provided the corresponding command is implemented in them.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

stream_control/2, stream_device/2, stream_property/2

Analyze an atom

sub_atom(+*Atom*, ?*StartLength*, ?*Length*, ?*RestLength*, ?*SubAtom* **)**

The predicate `sub_atom/5` unifies *SubAtom* with all substrings of *Atom* by backtracking. The predicate succeeds, if *SubAtom* is a part of *Atom*, beginning at the position *StartLength* and having the length *Length*.

Length is unified with the length of *SubAtom*, *StartLength* with the length of the substring to the left of *SubAtom* and *RestLength* with the length of the substring to the right of *SubAtom*.

When the arguments *StartLength*, *Length*, and/or *RestLength* are instantiated, arithmetic expressions will be evaluated.

Arguments

Atom	Atom
StartLength	Integer or integer expression, $StartLength \geq 0$
Length	Integer or integer expression, $Length \geq 0$
RestLength	Integer or integer expression, $RestLength \geq 0$
SubAtom	Atom

Exceptions

instantiation_error
The argument *Atom* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Atom* must be an atom, but is a term of another type.

type_error(atom)
The argument *SubAtom* must be a variable or an atom, but is a term of another type.

type_error(integer)
The argument *StartLength*, *Length* or *RestLength* must be a variable, an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *StartLength*, *Length* or *RestLength* must be a positive integer, but is negative.

Furthermore, if an arithmetic expression is specified for the argument *StartLength*, *Length* or *RestLength*, all the exceptions for `is/2` can occur.

Example

To obtain a substring:

```
[user] ?- sub_atom(substring,5,7,_,Z).<ENTER>

no
```

To split a string into substrings of length 3:

```
[user] ?- sub_atom(abcde,P,3,_,Z).<ENTER>

P       = 0
Z       = abc ;<ENTER>

P       = 1
Z       = bcd ;<ENTER>

P       = 2
Z       = cde ;<ENTER>

no
```

To obtain all substrings as of position 3:

```
[user] ?- sub_atom(abcde,3,_,_,Z).<ENTER>

Z       = '' ;<ENTER>

Z       = d ;<ENTER>

Z       = de ;<ENTER>

no
```

To split an atom fully into all possible substrings:

```
[user] ?- sub_atom(ab,_,_,_,Z).<ENTER>

Z       = '' ;<ENTER>

Z       = a ;<ENTER>

Z       = ab ;<ENTER>

Z       = '' ;<ENTER>

Z       = b ;<ENTER>
```

```
Z       = '' ;<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

The use of arithmetic expressions in the arguments *StartLength*, *Length* and *RestLength* is not allowed in the ISO standard for Prolog. In the ISO compatibility mode (see Prolog flag `iso`), therefore, no arithmetic expressions may be used as arguments for these predicates. Instead, an evaluation must be programmed explicitly with the help of `is/2` if necessary.

See also

atom_concat/3, atom_part/4, atom_prefix/3, atom_suffix/3, getchar/3, index/3, atom_split/3

Assignment of syntax error number to error message

syntax_error(+*Number*, ?*Message* **)**

The predicate `syntax_error/2` unifies *Message* with the syntax error message associated with *Number*.

If a *Number* for which no error message is known is specified, *Message* is unified with the atom `'unknown syntax error'`.

The following syntax errors may occur:

Number	Message
1	`.  expected`
2	`) expected`
3	`end of comment */ expected`
4	`illegal character`
5	`start of term expected`
6	`unknown stand alone character`
7	`operator expected`
8	`in/postfix operator or end of term expected`
9	`too large constant`
10	`| ] or , expected`
11	`} expected`
12	`' or " expected`
13	`, or ) expected`
14	`illegal number`
15	`precedence error`
16	`] expected`
17	`interrupt`
18	`i/o error`
19	`illegal escape sequence`
20	`illegal character code`
21	`exceeded max_arity`
22	`unbound variable expected`

Arguments

Number	Integer or integer expression
Message	Atom

Exceptions

instantiation_error
The argument *Number* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Number* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(atom)
The argument *Message* must be a variable or an atom, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

read_error/2/3, read/1/2, read_term/2/3

Start the operating system command interpreter

system

The predicate `system/0` starts the operating system command interpreter whose program name (path) is defined in the environment variable `SHELL`. This environment variable usually defines the login shell of a user and is set to a default value by the operating system.

If the environment variable is not set, the default command interpreter for the operating system (UNIX: Bourne shell) is started.

The predicate succeeds when the shell is exited. This makes it possible to work with the operating system interactively without quitting IF/Prolog.

Exceptions

system_error(command_failed)
A command passed to the operating system or a corresponding operating system function could not be executed successfully.

Hints

This predicate is system dependent and may not be available on all operating systems.

Example

```
[user] ?- system.<ENTER>
$ pwd<ENTER>
/usr1/pri
$ ls *.pro<ENTER>
widgets.pro
$ <EOF>
yes

[user] ?-
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

system/1/2/3/5, getenv/2

Execute an operating system command

system(+*Command* **)**

system(+*Command*, ?*ExitStatus* **)**

The predicate `system/1` executes *Command* in the operating system environment using `system(3C) in [10]`. After the specified command is completed, the predicate `system/1` succeeds; otherwise, if command execution is interrupted, an exception occurs.

The predicate `system/2` acts in the same way as `system/1` and unifies *ExitStatus* with the exit status of the executed command (see `halt/1`).

Arguments

Command	Atom
ExitStatus	Integer

Exceptions

instantiation_error
The argument *Command* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Command* must be an atom, but is a term of another type.

type_error(integer)
The argument *ExitStatus* must be a variable or an integer, but is a term of another type.

system_error(command_failed)
A command passed to the operating system or a corresponding operating system function could not be executed successfully.

Hints

This predicate is system dependent and may not be available on all operating systems.

The predicates `system/1/2` are based on the C function `system(3C)` (see [10]), i.e. *Command* may contain any file redirections, metacharacters for filename generation, etc., that are normally permitted for the command interpreter.

Example

```
[user] ?- system(date).<ENTER>
Thu Jul 14 09:46:07 DST 1994

yes

[user] ?- system('ls -l *.pro | fgrep example').<ENTER>
-rw-rw----   1 brc      prolog        19 Jul 14 09:45 example.pro

yes

[user] ?- system('cal 10 1996').<ENTER>
   October 1996
 S  M Tu  W Th  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31


yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

system/0/3/5, halt/0/1

Execute an operating system command with input and output specified

system(+*Command*, ?*Input*, ?*Output* **)**

system(+*Command*, ?*Input*, ?*Output*, ?*Error*, ?*Pid* **)**

The predicate `system/3` executes *Command* in an operating system shell using the specified I/O streams *Input* and *Output* as the standard input and standard output of the *Command*. This makes it possible to supply inputs to a command from IF/Prolog and to process outputs from a command within IF/Prolog.

If *Input* is a variable, the system creates and opens a pipe whose input direction becomes the standard input of *Command* and whose output direction is bound to this variable as a IF/Prolog output stream. All subsequent outputs on this output stream are sent to the specified *Command* as inputs.

If *Output* is a variable, the system creates and opens a pipe whose output direction becomes the standard output of *Command* and whose input direction is bound to this variable as a IF/Prolog input stream. All outputs from *Command* are then available as inputs on this input stream.

The predicate `system/5` is functionally equivalent to `system/3`, except that the standard error output can also be set for command execution and *Pid* is unified with the process ID number of the generated child process.

Arguments

Command	Atom
Input	Stream (system-defined term) or alias (atom)
Output	Stream (system-defined term) or alias (atom)
Error	Stream (system-defined term) or alias (atom)
Pid	Integer

Exceptions

instantiation_error
The argument *Command* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Command* must be an atom, but is a term of another type.

system_error(fork_failed)
No new process could be created. This is often due to resource bottlenecks or limitations in the operating system.

system_error(exec_failed)
The command *Command* could not be started. Possibly this is because the command could not be found in the current environment.

domain_error(stream_or_alias)
The argument *Input*, *Output* or *Error* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Input*, *Output* or *Error* must be associated with an open input/-output stream.

permission_error(input,stream)
The argument *Input* is not a variable or a stream which has been opened for input (open mode `read`).

permission_error(output,stream)
The argument *Output* or *Error* is not a variable or a stream which has been opened for output (open mode `write` or `append`).

permission_error(duplicate,stream)
The argument *Input*, *Output* or *Error* specifies an input/output stream that is not suitable for the required operation.

type_error(integer)
The argument *Pid* must be a variable or an integer, but is a term of another type.

Hints

These predicates are system dependent and may not be available on all operating systems.

Command is executed with `exec/1` in a separate process.

The completion of the command to be executed is awaited only if *Input* is associated with a terminal; otherwise, execution takes place asynchronously with the IF/Prolog process.

Extreme caution is required in connection with signals. For example, if a command which reads from `user_input` (a terminal) and ignores the signal `interrupt` is executed with `system/3`, the arrival of this signal may cause `system/3` to abort with an exception, while the command continues to exist as an independent process. The IF/Prolog system would return to the input loop `break/0` and wait for input from the user, but so would the command, so both processes would end up competing for the user input.

Example

The following example shows how output from a command is redirected to a file:

```
[user] ?- open(example,write,Stream),<ENTER>
          system('ls -l *.pro',user_input,Stream),<ENTER>
          close(Stream).<ENTER>

Stream  = '$stream'(7) <ENTER>

yes

[user] ?- system('cat example').<ENTER>

-rw-r-----  1 joey      1327 May 29 15:33 heidi.pro
-rw-r-----  1 tommy     2396 Dec 11 10:45 ramona.pro
-rw-r-----  1 marc      3923 Dec 11 10:45 sheena.pro

yes
```

User inputs are counted by the command `wc`, and the result is made available in IF/Prolog as a number:

```
[user] ?- system('wc -l;echo .',user_input,S),<ENTER>
          read(S,Number).<ENTER>
line_1<ENTER>
line_2<ENTER>
line_3<ENTER>
<EOF>

S       = '$stream'(6)
Number  = 3 <ENTER>

yes
```

The following predicate shows all Prolog source files which are present in the current directory. Two commands are connected using a pipe for this purpose:

```
[user] ?- listing.<ENTER>

% *** user: sources / 0 ***
sources :-
    system(ls,user_input,Pipe),
    system('fgrep .pro',Pipe,user_output),
    close(Pipe) .

yes

[user] ?- sources.<ENTER>

yes
```

`example.pro`

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

system/0/1/2, open/3/4, unix_fork/1, exec/1/2, unix_kill/2, unix_make_pipe/1, unix_wait/1/2, stream_control/2

Query machine identification

system_hostid(?*Hostid*)

The predicate `system_hostid/1` unifies *Hostid* with an integer representing a unique machine identification.

Arguments

Hostid	Integer

Exceptions

type_error(integer)
The argument *Hostid* must be a variable or an integer, but is a term of another type.

Hints

This host identification varies from one computer to another and can be used in an application to control actions specific to the machine.

Example

```
[user] ?- system_hostid(Hostid).<ENTER>

Hostid  = 1762018966 <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `system_hostid/1` is new.

See also

prolog_serial/1, prolog_version/1, proroot/1, system_name/1

Query operating system

system_name(?*SystemName*)

The predicate `system_name/1` unifies *SystemName* with an atom containing the name of the operating system.

Arguments

SystemName	Atom

Exceptions

type_error(atom)
The argument *SystemName* must be a variable or an atom, but is a term of another type.

Hints

This name varies from one operating system to another and can be used in an application to control actions specific to the operating system.

Example

```
[user] ?- system_name(Sys).<ENTER>

Sys     = 'SunOs 5.3' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

prolog_serial/1, prolog_version/1, proroot/1, system_hostid/1

Query system parameters from IF/Prolog call

system_parameters(?*ParameterList*)

The predicate `system_parameters/1` unifies *ParameterList* with a list of all parameters which were specified on calling IF/Prolog and identified as system parameters.

The first element of the list is the name under which IF/Prolog was called.

System parameters are all the call parameters which, at call time, appear in the command line before the first parameter that is not one of the parameters listed below, and which do not follow the special system parameter `--`.

Some system parameters require an argument. This argument must follow the system parameter in question as a separate command-line argument.

Arguments

ParameterList	List of atoms

Exceptions

type_error(list)
The argument *ParameterList* must be a variable or a list, but is a term of another type.

type_error(atom)
An element of *ParameterList* must be a variable or an atom, but is a term of another type.

Hints

The first element of *ParameterList* need not match the name given by the user on the command line; it may also be an appropriately expanded pathname under which IF/Prolog was found in the system. This name will usually vary from system to system.

Example

The file *example.pro* contains the following Prolog program:

```
program :-
        system_parameters(List),
        write(List),nl.
```

Calling IF/Prolog as shown below would then return:

```
$ prolog -nonotify -c example -data hello<ENTER>
IF/PROLOG V5.1
Copyright (C) Siemens Nixdorf Information Systems AG, 1996

[prolog,-nonotify,-c,example]
$
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

program_parameters/1, user_parameters/1

Move the cursor forward

tab(+*Number* **)**

tab(@*Stream*, +*Number* **)**

The predicate `tab/1` outputs *Number* spaces on the *current* output stream (`current_output`).

The predicate `tab/2` has the same effect, but on the specified output stream *Stream*.

If *Number* is less than 1 or negative, the predicates `tab/1/2` has no effect.

Arguments

Number	Integer or integer expression
Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* or *Number* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Number* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

Hints

The predicate `tab/1/2` positions the cursor relative to its current position; `outtab/1/2` positions it absolutely.

Example

```
[user] ?- write('Test'), tab(5),<ENTER>
          write('Prolog'), nl.<ENTER>
Test     Prolog

yes

[user] ?- write(not), tab(-5),<ENTER>
          write('effective'), nl.<ENTER>
noteffective

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

outtab/1/2, write/1/2, put_char/1/2

Set current output stream

tell(@*Stream*)

The predicate `tell/1` makes *Stream* the current output stream.

If the argument *Stream* is not an input/output stream and not a defined alias, it must be an atom. A file of this name is then opened for writing. If this succeeds, the name is simultaneously registered as an alias for the opened output stream and the stream becomes the current output stream.

Arguments

Stream	Stream (system-defined term), alias (atom) or filename (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

domain_error(file_name)
The value of the argument *Stream* is not a valid name for a file.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

permission_error(open,source_sink)
The argument *Stream* specifies a file (or a different I/O medium) that cannot be opened.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

Hints

All output predicates which do not have an explicit stream argument refer to the current output stream. This stream can always be addressed via the built-in alias `current_output` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

told/0, telling/1, set_output/1, error_tell/1, see/1, open/3/4, assign_alias/2

Query current output stream

telling(?*Stream*)

The predicate `telling/1` unifies *Stream* with the `current` output stream.

The current output stream may be a stream that was set with `tell/1` or `set_output/1`.

Arguments

Stream	Stream (system-defined term) or alias (atom)

Exceptions

domain_error(stream_or_alias)
: The argument *Stream* must be a variable or an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

Hints

The predicate `telling/1` can unify *Stream* with an alias or the filename that was possibly used with `tell/1` or `open/4`.

All output predicates which do not have an explicit stream argument refer to the current output stream. This stream can always be addressed via the built-in alias `current_output` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

tell/1, told/0, set_output/1, current_output/1, error_telling/1, seeing/1

Initiate a context jump

throw(@*Ball*)

The language construct `throw/1` is a special predicate that neither succeeds nor fails. Its effect is purely procedural and causes normal execution to be interrupted.

Calling `throw/1` results in an attempt to search for the innermost context that can intercept context jumps. Such a context may have been created with

`context(` *Goal*`, [..., catch(` *CatchMask* `) =>` *SubstituteGoal*`,...])`

or

`catch(` *Goal*`,` *CatchMask*`,` *SubstituteGoal* `)`

If the *CatchMask* defined here can be successfully unified with a copy of *Ball*, execution returns to the point at which the call was made, and the call to `catch/3` or `context/2` is replaced by the call to *SubstituteGoal*. The result of *SubstituteGoal* then becomes the result of `catch/3` or `context/2`. Backtracking by *SubstituteGoal* is possible. If the unification of *CatchMask* with a copy of *Ball* is not successful, the procedure described above is applied to the next outer context.

If no matching context can be found, program execution is aborted with an error message.

Arguments

Ball	Nonvar

Exceptions

instantiation_error
The argument *Ball* must not be a variable, but a variable was specified.

system_error(missing_catcher)
The argument *Ball* does not match any active context of the Prolog system.

Hints

The predicate `break/0` (Prolog input loop) defines a context which intercepts all context jumps.

Example

```
[user] ?- listing.<ENTER>

% *** user: foo / 1 ***
foo(X) :-
    Y is X * 2,
    throw(test(Y)) .

yes

[user] ?- catch( foo( 5 ), test( Y ), write( Y )).<ENTER>
10
Y       = 10 <ENTER>

yes
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

Compatibility

V5.0A The data structures used by the system in implicit `throw/1` calls have changed and are incompatible with the previous version of IF/Prolog.

V5.0B ISO conformance: exception is generated when *Ball* is a variable.

See also

abort/0, catch/3, context/2, exception/2

Determine time and time zones

timezone (+*Time*, **?***TimeZone*, **?***AlternateZone*, **?***DaylightSaving* **)**

The predicate `timezone/4` determines for a given system time (see function `time` in `is/2`) the differences in seconds between UTC (Universal Time Coordinated = Greenwich Mean Time) and local standard time (LST), resp. an alternate time zone, and if an alternate time zone (ATZ) is used (daylight savings time).

Arguments

Time	Integer
TimeZone	Integer, difference in seconds between UTC and LST
AlternateZone	Integer, difference in seconds between UTC and ATZ
DaylightSaving	Atom: on | off

Exceptions

instantiation_error
The argument *Time* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Time* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(integer)
The argument *TimeZone* or *AlternateZone* must be a variable or an integer, but is a term of another type.

type_error(atom)
The argument *DaylightSaving* must be an atom, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Time*, all the exceptions for `is/2` can occur.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1B The predicates `timezone/4` are new.

See also

is/2, local_time/9

Reset current output stream

told

The predicate `told/0` resets the default output stream `user_output` as the current output stream.

If the earlier current output stream was not a default stream and is not simultaneously the current error output stream, it is closed (see `close/1`).

Hints

All output predicates which do not have an explicit stream argument refer to the current output stream. This stream can always be addressed via the built-in alias `current_output` as well.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

tell/1, telling/1, current_output/1, set_output/1, close/1/2

Activate tracing for a goal

trace(+*Goal* **)** `[` **@** +*Module* `]`

The predicate `trace/1` activates the tracer for executing *Goal*.

Before *Goal* is executed, the user-defined predicate `trace_begin/0` is called, if it is visible in the calling module or in the specified *Module*.

After *Goal* has been executed, the user-defined predicate `trace_end/0` is called, if it is visible in the calling module or in the specified *Module*.

Arguments

Goal	Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

If `trace/1` is called within `trace/1` or `debug/1`, this call is ignored.

The predicate `trace/1` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in

the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

call/1, trace_begin/0, trace_end/0, debug/1, trace_output/4, set_prolog_flag/2

User-defined tracer initialization

trace_begin `[ :-` *Body* `]`

The predicate `trace_begin/0` is not a built-in predicate but one which can be defined by the user. It is called in the module in which `trace/1` was called.

It is called automatically by `trace/1` when tracing is started.

The user can define `trace_begin/0` to perform preliminary activities, such as defining default breakpoints with `trace_config/3`, when the tracer is started.

Hints

Nested calls involving `debug/1` and/or `trace/1` are ignored. If `trace/1` is called several times during a goal execution, `trace_begin/0` is called each time.

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

trace/1, trace_end/0, trace_config/3

Configure the tracer

trace_config(+*Parameter*, ?*Old*, +*New* **)**

The predicate `trace_config/3` is used to query and modify trace parameters.

Parameter is used to specify the trace parameter to be queried or modified. At the time of the call, *Old* is unified with the previous value of the trace parameter and then, if unification was successful, the new value *New* is set. The following tracer parameters are available (default values are printed in boldface):

Parameter	Value	Description
`call`	**on**	The tracer logs every pass through this port or calls the user-defined predicate `trace_output/4`.
	`off`	The tracer does not log passes through this port.
`exit`	**on**/`off`	the same applies to the `EXIT` port
`redo`	**on**/`off`	the same applies to the `REDO` port
`fail`	**on**/`off`	the same applies to the `FAIL` port
`error`	**on**/`off`	the same applies to the `ERROR` port
`wakeup`	**on**/`off`	the same applies to the `WAKEUP` port
`suspend`	**on**/`off`	the same applies to the `SUSPEND` port
`trymatch`	`on`/**off**	the same applies to the `TRYMATCH` port
`failmatch`	`on`/**off**	the same applies to the `FAILMATCH` port
`enterbody`	**on**/`off`	the same applies to the `ENTERBODY` port
`exitbody`	**on**/`off`	the same applies to the `EXITBODY` port
`redobody`	`on`/**off**	the same applies to the `REDOBODY` port
`failbody`	`on`/**off**	the same applies to the `FAILBODY` port
`display_depth`	**7**	The maximum structure depth is restricted by this parameter when terms are output by the tracer. If `0` is specified, there is no restriction.
`trace_length`	**100**	Number of trace outputs to be stored if the alpha interface is being used.

Arguments

Parameter	Atom, see table above
Old	Term, context-dependent
New	Term, context-dependent

Exceptions

instantiation_error
The argument *Parameter* or *New* or a subterm must not be a variable, but a variable was specified.

type_error(atom)
The argument *Parameter* must be an atom, but is a term of another type.

domain_error(config_option)
The argument *Parameter* is not a valid configuration parameter.

type_error(integer)
The argument *New* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *New* must be a positive integer, but is negative.

type_error(atom)
The argument *New* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *New* must be one of the atoms `on` or `off`, but is another atom.

Furthermore, if an arithmetic expression is specified for the argument *New*, all the exceptions for `is/2` can occur.

Hints

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

trace/1, debug_config/3

User-defined termination of the tracer

trace_end `[ :-` *Body* `]`

The predicate `trace_end/0` is not a built-in predicate but one which can be defined by the user. It is called in the module in which `trace/1` was called.

It is called automatically after `trace/1` has been processed.

The user can define `trace_end/0` to perform terminating tasks, such as closing files, when the tracer is deactivated.

Hints

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

trace/1, trace_begin/0

User-defined output predicate for the tracer

trace_output(@*CallNumber*, +*Port*, @*Goal*, @*VariableNames*) `[ :-` *Body* `]`

The predicate `trace_output/4` is not a built-in predicate but one which can be defined by the user. It is called in the module in which `trace/1` was called.

It is automatically called for the output of the trace log instead of the internal output function of the tracer. This enables the user to create a separate log or to process data from the tracer directly.

The predicate `trace_output/4` is called by the tracer with *CallNumber* as a list with the format `[`*RecursionDepth*,*SubGoalNumber*`]`, the name of the specified *Port* and the *Goal* to be traced.

If an exception has been triggered, the predicate `trace_output/4` is called with *Port* as a null atom `''` and *Goal* as the term `error`(*Error*, *Info*). The argument *VariableNames* contains the empty list.

Arguments

CallNumber	List of integers
Port	Atom: call | exit | fail | redo | suspend | wakeup | error | failmatch | enterbody | exitbody | redobody | trymatch | failbody | ”
Goal	Term
VariableNames	List with the format `[`*Name* = *Variable*, ...`]`
Name	Atom
Variable	Variable

Hints

This predicate is not part of the IF/Prolog runtime system and is therefore not available for applications intended for third parties.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

trace/1

Goal which is always true

true

The language construct `true/0` succeeds exactly once.

Hints

The language construct `true/0` is **not** a predicate, so it cannot be redefined. The predicates `current_predicate/1` and `current_visible/2` do not indicate language constructs. Explicit module qualification with `@/2` or `:/2` is not permitted for language constructs (with the exception of `call/1`).

Standard

This language construct complies with the definition of the ISO standard for Prolog.

See also

fail/0

Query the screen size

tty_size(?*Lines*, ?*Columns* **)**

tty_size(@*Stream*, ?*Lines*, ?*Columns* **)**

The predicate `tty_size/2` queries the size of the screen associated with the current output stream.

If there is no screen associated with the current output stream or if the device driver for the stream is not able to supply this information, *Lines* and *Columns* are unified with 0.

The predicate `tty_size/3` acts in the same way as `tty_size/2`, but uses the screen associated with the specified output stream *Stream*.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Lines	Integer
Columns	Integer

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

type_error(integer)
The argument *Lines* or *Columns* must be a variable or an integer, but is a term of another type.

Hints

The terminal control predicates are implemented via the appropriate stream control functions of the device drivers (see `stream_control/2`).

The standard device drivers implement screen control via the **terminfo/termcap** service of the operating system. They support only one terminal type for all output streams.

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

move_cursor/2/3, get_last/2/3

Unify terms with occurs check

unify_with_occurs_check(*?Term1*, *?Term2* **)**

The predicate `unify_with_occurs_check/2` unifies *Term1* with *Term2* after performing an occurs check.

This means that the system first checks whether either of the terms *Term1* or *Term2* contains the other as a subterm. If this is true, the unification fails.

This check is performed in each step of the unification.

Arguments

Term1	Term
Term2	Term

Example

A variable cannot be unified with a structure that contains the same variable:

```
[user] ?- unify_with_occurs_check(X+1,X).<ENTER>

no

[user] ?- unify_with_occurs_check(1+X,X).<ENTER>

no

[user] ?- unify_with_occurs_check(f(X),f(g(X))).<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

=/2, cyclic/1

Spawn process

unix_fork(?*Pid*)

The predicate `unix_fork/1` duplicates the current IF/Prolog process by creating a new process which is the child of the current IF/Prolog process and which runs IF/Prolog with the same processing status as its parent.

The predicate succeeds in both processes, except that *Pid* is unified with the number of the child process in the parent process and with 0 in the child process.

Arguments

Pid	Integer

Exceptions

system_error(fork_failed)
No new process could be created. This is often due to resource bottlenecks or limitations in the operating system.

type_error(integer)
The argument *Pid* must be a variable or an integer, but is a term of another type.

Hints

The distinction of the return value for `Pid` can be subsequently used to enter different program branches in parent- and child-process. The duplicated IF/Prolog system in the child process can also be overlaid with some other program using `exec/1/2`.

The distinction could also be used to try out changes in the database without affecting the original IF/Prolog system.

The predicate `unix_fork/1` is based on the C function `fork(2)` (see [10]).

This predicate is system dependent and may not be available on all operating systems.

Example

The following predicate creates a genuine duplicate of the IF/Prolog system. The second input loop thus operates in a child process:

```
[user] ?- listing.<ENTER>

% *** user: test / 0 ***
```

```
test :-
    unix_fork(Pid),
    Pid =\= 0,
    repeat,
    unix_wait(Pid),
    ! .
test :-
    break,
    halt .

yes
```

On exiting this input loop, the user is returned to the original process in its original unaltered state:

```
[user] ?- test.<ENTER>

[user][2] ?- asserta(a).<ENTER>

yes

[user][2] ?- listing.<ENTER>

% *** user: a / 0 ***
a.

% *** user: test / 0 ***
test :-
    unix_fork(Pid),
    Pid =\= 0,
    !,
    repeat,
    unix_wait(Pid),
    ! .
test :-
    program,
    halt .

yes

[user][2] ?- halt.<ENTER>

yes

[user] ?- listing.<ENTER>
```

```
% *** user: test / 0 ***
test :-
    unix_fork(Pid),
    Pid =\= 0,
    !,
    repeat,
    unix_wait(Pid),
    ! .
test :-
    program,
    halt .

yes
```

See also the example in the section *Process management and communication* in the *IF/Prolog User's Guide* [2].

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

exec/1/2, unix_wait/1/2, unix_kill/2, system/0/1/2/3/5

Query process ID of current process

unix_getpid(?*Pid*)

The predicate `unix_getpid/1` unifies *Pid* with the process ID number of the IF/Prolog process from which it was called.

Arguments

Pid	Integer

Exceptions

type_error(integer)
The argument *Pid* must be a variable or an integer, but is a term of another type.

Hints

The predicate `unix_getpid/1` is based on the C function `getpid(2)` (see [10]).

This predicate is system dependent and may not be available on all operating systems.

Example

```
[user] ?- unix_getpid(Pid).<ENTER>

Pid     = 3435 <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

unix_kill/2, unix_wait/1/2, unix_fork/1

Send signal to a process

unix_kill(+*Pid*, +*Signal* **)**

The predicate `unix_kill/2` sends the signal specified by *Signal* to the process specified by *Pid*. If the signal in question cannot be sent, `unix_kill/2` fails.

If 0 is specified for *Signal*, no signal is sent, but a check is made to determine whether a signal could be sent to *Pid*. This feature can be used to check whether a particular process exists. If the process does not exist, `unix_kill/2` fails.

Some of the more important signal numbers are listed below:

Prolog name	C name	Signal number
`-`	SIGHUP	1
`interrupt`	SIGINT	2
`quit`	SIGQUIT	3
`abort`	SIGABRT	6
`-`	SIGKILL	9
`pipe`	SIGPIPE	13
`alarm`	SIGALRM	14
`termination`	SIGTERM	15
`user_1`	SIGUSR1	16
`user_2`	SIGUSR2	17

If a process ID number greater than zero is specified for *Pid*, the signal is sent to precisely that process. If 0 is specified for *Pid*, the signal is sent to all processes that belong to the same process group as the current Prolog process. Other special interpretations of *Pid* can be obtained from the description of the C function `kill(2)` (see [10]).

Arguments

Pid	Integer or integer expression
Signal	Atom, integer or integer expression

Exceptions

instantiation_error
: The argument *Pid* or *Signal* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
: The argument *Pid* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

type_error(atom_or_integer)
The argument *Signal* must be an atom, an integer or an integer expression, but is a term of another type.

domain_error(signal_name)
The argument *Signal* must be one of the built-in signal names, but is another atom.

Furthermore, if an arithmetic expression is specified for the argument *Pid* or *Signal*, all the exceptions for `is/2` can occur.

Hints

The predicate `unix_kill/2` is based on the C function `kill(2)` (see [10]).

This predicate is system dependent and may not be available on all operating systems.

Example

```
[user] ?- unix_getpid(Pid),<ENTER>
          unix_kill(Pid,interrupt),<ENTER>
          write(hello).<ENTER>

*** E X C E P T I O N: system_error
>>> error = interrupt
>>> signal = interrupt
>>> interrupted = system : unix_kill(3443,interrupt)
>>> goal = true

no

[user] ?- unix_kill(9999,0).<ENTER>

no
```

See also the example in the section *Process management and communication* in the *IF/Prolog User's Guide* [2].

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

unix_fork/1, unix_wait/1/2, unix_getpid/1, signal_control/2, signal_handler/3

Create a pipe

unix_make_pipe(-*Pipename*)

The predicate `unix_make_pipe/1` creates a pipe and unifies *Pipename* with a medium name that can be directly specified as the first argument of `open/3/4`. (`unix_fork/1`, `exec/1/2`).

Arguments

Pipename	Name of the pipe medium

Exceptions

type_error(variable)
The argument *Pipename* must be a variable, but is a term of another type.

system_error(could_not_create_pipe)
The operating system was not able to create a pipe. This is often due to resource bottlenecks or limitations in the operating system.

Hints

Pipes are normally used for process communication. They are inherited by child processes

If a pipe that is created with `unix_make_pipe/1` is opened in IF/Prolog (with `open/3/4`) for reading or writing, the redundant side of the pipe is closed automatically.

The predicate `unix_make_pipe/1` is based on the C function `pipe(2)` (see [10]).

This predicate is system dependent and may not be available on all operating systems.

Example

See the example in the section *Process management and communication* in the *IF/Prolog User's Guide* [2].

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

open/3/4, close/1/2, unix_fork/1, exec/1/2

Wait for termination of a child process

unix_wait(?*Pid*)

unix_wait(?*Pid*, ?*ExitStatus*)

The predicate `unix_wait/1` suspends the current IF/Prolog process until a direct child process terminates, and then unifies *Pid* with the process ID number of that process.

In some circumstances, incoming signals may interrupt this wait state as well, even if such signals are handled by IF/Prolog or the user. *Pid* is unified with `-1` in such cases.

The predicate `unix_wait/2` acts in the same way as `unix_wait/1`, except that *ExitStatus* is additionally unified with the the exit status of the terminated child process.

Arguments

Pid	Integer
ExitStatus	Integer

Exceptions

system_error(no_child)
The predicate `unix_wait/1/2` cannot be executed since no child process exists.

type_error(integer)
The argument *Pid* or *ExitStatus* must be a variable or an integer, but is a term of another type.

Hints

The predicates `unix_wait/1/2` are based on the C function `wait(2)` (see [10]).

This predicate is system dependent and may not be available on all operating systems.

Example

The following predicate implements `system/2`:

```
[user] ?- listing.<ENTER>

% *** user: my_system / 2 ***
my_system(_,Stat) :-
    unix_fork(Pid),
```

```
    Pid =\= 0,
    !,
    repeat,
    unix_wait(Pid,Res),
    !,
    Res = Stat .
my_system(Com,_) :-
    unix_exec(Com) .

yes

[user] ?- my_system('exit 7',End).<ENTER>

End     = 7 <ENTER>

yes
```

See also the example in the section *Process management and communication* in the *IF/Prolog User's Guide* [2].

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

unix_fork/1, unix_getpid/1, unix_kill/2

Remove module from the database

unload(+*Module* **)**

The predicate `unload/1` removes all information in connection with the specified *Module* from the database. If the specified *Module* was the current module, the built-in module `user` becomes the current module.

The *Module* will then have neither an interface nor a body, but its name will still be known (see `current_module/1`). The built-in module `user` is an exception to this rule. If this module is deleted with `unload/1`, an empty module interface and an empty module body is created immediately. Other built-in modules cannot be removed with `unload/1`.

Arguments

Module	Atom, name of a module

Exceptions

instantiation_error
The argument *Module* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

existence_error(module)
The argument *Module* is not the name of an existing module.

permission_error(access,system_module)
The predicate `unload/1` must not be executed for a system module.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

compile/1, load/1, module/1, current_default_module/1, current_module/1

Delete a global variable

unset_global(+*Name* **)** `[` **@** +*Module* `]`

The predicate `unset_global/1` deletes the global variable *Name* in the calling module or in the specified *Module*.

The predicate `unset_global/1` will succeed even if no global variable called *Name* exists.

Arguments

Name	Atom

Exceptions

instantiation_error
The argument *Name* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Name* must be an atom, but is a term of another type.

Hints

Global variables are always assigned to a module; they can be neither exported nor imported. In operations with global variables, therefore, only those global variables defined in the calling module or in the specified *Module* are taken into account.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

current_global/1, get_global/2, pop_global/2, push_global/2, set_global/2

Query user parameters from IF/Prolog call

user_parameters(?*ParameterList*)

The predicate `user_parameters/1` unifies *ParameterList* with a list of all user parameters which were specified on calling IF/Prolog.

User parameters are all the call parameters which are not system parameters (see `system_parameters/1`) or which are entered on the command line after the special system parameter `--`.

Arguments

ParameterList List of atoms

Exceptions

type_error(list)
The argument *ParameterList* must be a variable or a list, but is a term of another type.

type_error(atom)
An element of *ParameterList* must be a variable or an atom, but is a term of another type.

Example

The file *example.pro* contains the following Prolog program:

```
program :-
        user_parameters(List),
        write(List),nl.
```

Calling IF/Prolog as shown below would then return:

```
$ prolog -c example -- -debug -data hello<ENTER>
IF/PROLOG V5.1
Copyright (C) Siemens Nixdorf Information Systems AG, 1996

*** consult 'example.pro': loaded in 0.02 sec.
[-debug,-data,hello]
$
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

program_parameters/1, system_parameters/1

Test for variable

var(@*TestTerm* **)**

The predicate `var/1` succeeds if *TestTerm* is an uninstantiated variable, otherwise it fails.

Arguments

TestTerm	Term

Example

```
[user] ?- var( X ).<ENTER>

X       = _68 <ENTER>

yes

[user] ?- var(atom).<ENTER>

no

[user] ?- X is 3 - 2, var( X ).<ENTER>

no
```

Standard

This predicate complies with the definition of the ISO standard for Prolog.

See also

atom/1, atomic/1, compound/1, cyclic/1, float/1, ground/1, integer/1, nonvar/1, number/1, rational/1, var/1

Change drive

windows_chdrive(`+`*Drivename* **)**

The predicate `windows_chdrive/1` changes the current drive to *Drivename*. An error occurs if the drive does not exist.

Arguments

Drivename	Letter, name of a drive

Exceptions

instantiation_error
The argument *Drivename* must not be a variable, but a variable was specified.

type_error(character)
The argument *Drivename* must be a character, but is a term of another type.

domain_error(drive_name)
The value of the argument *Drivename* is not a valid drive character.

system_error(chdrive_failed)
The operating system did not allow to change to the requested drive.

Hints

To write code that should run under diverse operating systems you can use the predicate `system_name/1` to determine the current operating system. On operating systems that do not support the concept of drives this predicate has no effect.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `windows_chdrive/1` is new.

Windows This predicate is only available under Windows and OS/2 operating systems.

See also

windows_getdrive/1, system_name/1, chdir/2, getcwd/2

Query drive

windows_getdrive(?*Drivename*)

The predicate `windows_getdrive/1` retrieves the name of the current drive and unifies it with *Drivename*.

Arguments

Drivename	Letter, name of a drive

Exceptions

type_error(character)
The argument *Drivename* must be a character, but is a term of another type.

domain_error(drive_name)
The value of the argument *Drivename* is not a valid drive character.

Hints

To write code that should run under diverse operating systems you can use the predicate `system_name/1` to determine the current operating system. On operating systems that do not support the concept of drives this predicate is not available.

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `windows_getdrive/1` is new.

Windows This predicate is only available under Windows and OS/2 operating systems.

See also

windows_chdrive/1, system_name/1, chdir/1, getcwd/1

Output a term

write(@*Term* **)**

write(@*Stream*, @*Term* **)**

The predicate `write/1` outputs *Term* on the current output stream. Existing operator definitions are taken into account.

The predicate `write/2` acts in the same way as `write/1`, but writes to the specified output stream *Stream*.

Both predicates output special characters from the term without special treatment.

Arguments

Term	Term
Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Hints

The call

```
write(Term)
```

is identical to

```
write_term(Term,[quoted(false),ignore_ops(false),numbervars(true)]).
```

The output produced by `write/1/2` is generally not suitable for re-input (e.g. with `read_term/2/3`).

Example

```
[user] ?- write(hello),nl.<ENTER>
hello

yes

[user] ?- write('new\nLine'),nl.<ENTER>
new
Line

yes

[user] ?- write('Robert\'s daughter'),nl.<ENTER>
Robert's daughter

yes

[user] ?- write('+'(3,'*'(7,8.3))),nl.<ENTER>
3 + 7 * 8.3

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

See also

writeq/1/2, write_term/2/3, write_canonical/1/2, read_term/2/3

Convert a term into an atom

write_atom(@*Term*, ?*Atom*)

The predicate `write_atom/2` converts *Term* into an atom, taking the operator definitions into account, and unifies the result with *Atom*.

Conversion to an atom occurs as with `write/1`, i.e. special characters in atoms are not treated specially in the output.

Arguments

Term	Term
Atom	Atom

Exceptions

type_error(atom)
The argument *Atom* must be a variable or an atom, but is a term of another type.

Hints

The call

```
write_atom(Term,Atom)
```

is identical to

```
open(string(test),write,S),
write(S,Term),
stream_control(S,string(Atom)),
close(S) .
```

An atom created by `write_atom/2` is generally not suitable for re-input (e.g. with `parse_atom/6`).

Example

```
[user] ?- write_atom(a(b+c),A).<ENTER>

A       = 'a(b + c)' <ENTER>

yes
```

```
[user] ?- write_atom(['*b','\'',3.0],A).<ENTER>

A       = '[*b,'',3]' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

writeq_atom/2, write/1/2, write_term/2/3, parse_atom/6, stream_control/2

Output a term

write_canonical(@*Term*)

write_canonical(@*Stream*, @*Term*)

The predicate `write_canonical/1` writes *Term* to the current output stream in a format that allows the term to be read in again.

The predicate `write_canonical/2` acts in the same way as `write_canonical/1`, but writes to the specified output stream *Stream*.

Operator notation is not used in the output, and atoms containing special characters are always quoted. All floating-point numbers that appear in *Term* are output with the highest possible precision.

Arguments

Term	Term
Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Hints

The call

```
write_canonical(Term)
```

is identical to

```
write_term(Term,[quoted(true),ignore_ops(true),numbervars(false)]) .
```

The output produced by `write_canonical/1/2` is suitable for re-input with `read_term/2/3`.

Example

```
[user] ?- write_canonical(a*4+7),nl.<ENTER>
+(*(a,4),7)

yes

[user] ?- write_canonical(['a\'b',3.7,17]),nl.<ENTER>
'.'('a''b','.'(3.7,'.'(17,[])))

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

See also

print/1/2, write/1/2, writeq/1/2, write_term/2/3

Formatted output of terms

write_formatted(+*Format*, @*TermList* **)**

write_formatted(@*Stream*, +*Format*, @*TermList* **)**

The predicate `write_formatted/2` outputs the terms specified in *TermList* to the current output stream in accordance with the formatting instructions in *Format*, while taking the existing operator definitions in the calling module or in the specified *Module* into account.

The predicate `write_formatted/3` acts in the same way as `write_formatted/2`, but writes to the specified output stream *Stream*.

The specification of *Format* is analogous to the C function `printf(3C)` (see [10]) with some Prolog-specific enhancements. The argument *Format* can be a string of any characters and formatting sequences, where the formatting sequences define how the associated elements of *TermList* are to be output, and the other characters are output without special treatment (as with `write/1/2`).

Each formatting sequence in *Format* is associated with exactly one argument in *TermList* in the order of its occurrence. The number of terms in *TermList* must thus match the number of conversion sequences in *Format* precisely, and the types of the associated terms must match the conversion sequences.

Conversion sequences are introduced with the character `'%'`. The specification `%%` is not a conversion sequence, but stands for the character `'%'` itself.

The general structure of a conversion sequence is as follows:

`%[`*Flags*`...][`*Field-length*`][.`*Precision*`]`*Type*

Except for *Type*, all other entries may be omitted. Several *Flags* may be specified.

The following *Flags* are available:

`+` Prints the sign with numbers in all cases.

`-` Left-justified output within the *Field-length*.

`' '` Prints a space instead of the sign for positive numbers.

`#` Canonical output; the significance is explained under the individual argument types.

`0` Replaces leading spaces by 0 in the output of integers.

The *Field-length* specifies the minimum field length required for the output of the associated argument. Output is normally right-justified in this field.

Precision defines the output precision of the corresponding argument in relation to the type of format. The significance for specific format types is given below:

`e/E` Number of significant positions.

`f` Number of decimal places.

`s` Maximum number of characters to be output.

`t/q` Maximum term depth to be output.

The individual format types are as follows:

`d` Prints an integer in decimal notation. The associated argument in *TermList* must be an integer or an integer expression.

`o` Prints an integer in octal notation. The associated argument in *TermList* must be an integer or an integer expression.

In canonical output (flag `#`), the number is preceded by 0.

`x` Prints an integer in hexadecimal notation. The associated argument in *TermList* must be an integer or an integer expression.

In canonical output (flag `#`), the number is preceded by `0x`.

`X` Prints an integer in hexadecimal notation. The associated argument in *TermList* must be an integer or an integer expression.

In canonical output (flag `#`), the number is preceded by `0X`.

`f` Prints a floating-point number in fixed-point notation. The associated argument in *TermList* must be an floating-point number or an expression.

`e` Prints a floating-point number in floating-point notation. The associated argument in *TermList* must be an floating-point number or an expression. The exponent part is introduced with the character `e`.

In canonical output (flag `#`), the number is represented with maximum precision.

`E` Prints a floating-point number in floating-point notation. The associated argument in *TermList* must be an floating-point number or an expression. The exponent part is introduced with the character `E`.

In canonical output (flag `#`), the number is represented with maximum precision.

`g` This type is equivalent to `f` or `e` and selects the shorter of the two notations.

`G` This type is equivalent to `f` or `E` and selects the shorter of the two notations.

`c` Prints a character. The associated argument in *TermList* must be a character.

`s` Prints a string. The associated argument in *TermList* must be an atom.

`n` No output. The associated argument in *TermList* should be a variable. This argument is unified with the number of characters that have been output by `write_formatted/2/3` so far. If the unification does not succeed, output is aborted, and the predicate `write_formatted/2/3` fails.

`t` Outputs a term in the same way as `write/1/2`.

`q` Outputs a term in the same way as `writeq/1/2`.

`w` Outputs a term in the same way as `writeq/1/2`, but using the maximum depth set for the Prolog flag `write_depth`.

Canonical output (flag `#`) of the term is identical to `write_canonical/1/2`.

Arguments

Format	Atom
TermList	List of terms
Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream*, *Format* or *TermList* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

type_error(atom)
The argument *Format* must be an atom, but is a term of another type.

domain_error(write_format)
The argument *Format* contains an invalid format specification.

type_error(list)
The argument *TermList* must be a list, but is a term of another type.

type_error(...)
An element of *TermList* must be of the specified type, but is a term of another type.

Furthermore, if an arithmetic expression is specified for an element in *TermList*, all the exceptions for `is/2` can occur.

Hints

Since the terms in *TermList* are processed in the order in which they are specified, partial output may occur even if *Format* or *TermList* contain errors.

The output produced by `write_formatted/1/2` is generally not suitable for re-input with `read_term/2/3`.

Example

```
[user] ?- write_formatted(<ENTER>
          'Result: %s = %t\n',<ENTER>
          ['Name',7.33]).<ENTER>
Result: Name = 7.33

yes
```

Standard

These predicates are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The exception `domain_error(print_format)` has been changed to `domain_error(write_format)` for more orthogonality.

See also

writeq/1/2, write_term/2/3, write_canonical/1/2

Formatted output of terms

write_formatted_atom(?*Atom*, +*Format*, @*TermList*)

The predicate `write_formatted_atom/3` outputs the terms specified in *TermList* to a stream of type `string` in accordance with the formatting instructions in *Format*. The contents of the string are then unified with *Atom*.

The predicate `write_formatted_atom/3` is a convenience predicate for using `write_formatted/3` with an output stream of type `string`. Please consult the manual entry for `write_formatted/2/3` for further information.

Arguments

Atom	Atom
Format	Atom
TermList	List of terms

Exceptions

type_error(atom)
The argument *Atom* must be a variable or an atom, but is a term of another type.

See also `write_formatted/2/3`.

Hints

The predicate can be implemented as follows:

```
write_formatted_atom(Atom, Format, ArgListe) :-
        open(string(dummy), write, S),
        write_formatted(S, Format, ArgListe),
        stream_control(S, string(Atom)),
        close(S).
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

Compatibility

V5.1A The predicate `write_formatted_atom/3` is new.

See also

writeq_atom/1/2, write_atom/2, write_formatted/2/3

Output a term

write_term(@*Term*, @*Options* **)** `[` @ +*Module* `]`

write_term(@*Stream*, @*Term*, @*Options* **)** `[` @ +*Module* `]`

The predicate `write_term/2` outputs *Term* to the current output stream in a format determined by the specified *Options*.

The predicate `write_term/3` acts in the same way as `write_term/2`, but writes *Term* to the specified output stream *Stream*.

The following terms can occur as elements of the *Options* list:

`maxdepth(+`*N*`)`
: Limits the structure depth for output to *N*. Any subterms in a structure at a structure depth greater than *N* are represented by the atom `'*'`. Structure levels are counted starting with 0.

  Lists are also structures, i.e. *N* limits the length of lists as well. In the case of lists, all elements that follow element *N* in the list (counting begins with 0) are replaced by the atom `'...'`.

`maxdepth(+`*N*`,+`*Atom1*`,+`*Atom2*`)`
: Limits the structure depth for output to *N*. Any subterms in a structure at a structure depth greater than *N* are represented by the atom *Atom1*. Structure levels are counted starting with 0.

  Lists are also structures, i.e. *N* limits the length of lists as well. In the case of lists, all elements that follow element *N* in the list (counting begins with 0) are replaced by the atom *Atom2*.

`quoted(+`*Bool*`)`
: This specification controls the behavior of `write_term/2/3` for atoms which contain special characters and for floating-point numbers.

  If *Bool* is `true`, atoms are output in accordance with Prolog syntax rules. For example, atoms containing special characters would be quoted. floating-point numbers are output with the highest possible precision.

  If *Bool* is `false` (default), the characters in such atoms are output without special treatment. floating-point numbers are output with the current floating-point format (see `float_format/2`).

`ignore_ops(+`*Bool*`)`
: This specification controls the behavior of `write_term/2/3` with regard to the use of operators.

  If *Bool* is `true`, operator definitions are ignored, and structures are always represented in normal structure notation.

If *Bool* is `false` (default), the existing operator definitions are used to represent output.

`portrayed(+`*Bool*`)`

If *Bool* is `true`, an attempt is first made to call the user-defined predicate `portray/2` in the calling module or in the specified *Module* in order to output *Term*. If this call succeeds, `write_term/2/3` will not generate its own output. If the call fails, the attempt is repeated for each subterm of *Term*, and the predicate `write_term/2/3` outputs those parts of *Term* for which the calls to `portray/2` fail.

If *Bool* is `false` (default), no attempt to call a user-defined predicate is made.

`numbervars(+`*Bool*`)`

If *Bool* is `true` and *Term* contains subterms in the form:

`'$VAR'(`*N*`),`

these subterms are replaced by symbolic variable names by constructing names in relation to the value of *N*. The numbers 0...25 are mapped to the names A...Z; the numbers 26...51 to the names A1...Z1, and so on.

`varnames(+`*VarList*`)`

If *Term* contains a variable, *VarList* is checked to see if it contains an element of the form

Name `=` *Variable*

for precisely that variable. If it does, the variable in question is not be represented by its internal name, but by the atom *Name* from *VarList*.

Arguments

Stream	Stream (system-defined term) or alias (atom)
Term	Term
Options	List of structures
N	Integer ≥ 0
Atom1, *Atom2*	Atom
Bool	Atom: true `|` false
VarList	List of the form `[`*Name* = *Variable*,...`]`
Name	Atom
Variable	Variable

Exceptions

instantiation_error

The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)

The argument *Stream* must be an input/output stream supplied by `open/3/4` or

an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

instantiation_error
The argument *Options* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *Options* must be a list, but is a term of another type.

domain_error(write_option)
An element of *Options* specifies an option that is not permitted.

Example

```
[user] ?- write_term(<ENTER>
            [struc(f(g)),one,two],<ENTER>
            [maxdepth(1)]),nl.<ENTER>
[struc(f(*)),one,...]

yes

[user] ?- write_term(<ENTER>
            [struc(f(g)),one,two],<ENTER>
            [maxdepth(1,stop,etc)]),nl.<ENTER>
[struc(f(stop)),one,etc]

yes

[user] ?- write_term('new\nline',[]),nl.<ENTER>
new
line

yes

[user] ?- write_term('new\nline',<ENTER>
```

```
            [quoted(true)]),nl.<ENTER>
'new\nline'

yes

[user] ?- write_term(a*b+c,[]),nl.<ENTER>
a * b + c

yes

[user] ?- write_term(a*b+c,<ENTER>
            [ignore_ops(true)]),nl.<ENTER>
+(*(a,b),c)

yes

[user] ?- write_term(f(X,Y),[]),nl.<ENTER>
f(_68,_69)

X       = _68
Y       = _69 <ENTER>

yes

[user] ?- write_term(f(X,Y),<ENTER>
            [varnames(['X'=X,'Y'=Y])]),nl.<ENTER>
f(X,Y)

X       = _68
Y       = _69 <ENTER>

yes

[user] ?- write_term(['$VAR'(0),'$VAR'(54)],<ENTER>
            [numbervars(true)]),nl.<ENTER>
[A,C2]

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

The options `maxdepth(...)`, `portrayed(...)` and `varnames(...)` are not required by the ISO standard for Prolog. Therefore, they are not available in ISO compatibility mode (see Prolog flag `iso`).

See also

print/1/2, write/1/2, writeq/1/2, write_canonical/1/2

Output a term in readable form

writeq(@*Term*)

writeq(@*Stream*, @*Term*)

The predicate `writeq/1` outputs *Term* on the current output stream. Existing operator definitions are taken into account.

The predicate `writeq/2` acts in the same way as `writeq/1`, but writes to the specified output stream *Stream*.

Both predicates output atoms with special characters in accordance with Prolog syntax, so these atoms can be read in again. Floating point numbers are output wich maximum precision.

Arguments

Term	Term
Stream	Stream (system-defined term) or alias (atom)

Exceptions

instantiation_error
The argument *Stream* must not be a variable, but a variable was specified.

domain_error(stream_or_alias)
The argument *Stream* must be an input/output stream supplied by `open/3/4` or an atom which has been registered with `assign_alias/2` or with the alias option for `open/4` as an alias for an input/output stream.

existence_error(stream)
The argument *Stream* must be associated with an open input/output stream.

permission_error(output,stream)
The argument *Stream* is not a stream which has been opened for output (open mode `write` or `append`).

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *Stream*.

Hints

The call

```
writeq(Term)
```

is identical to

```
write_term(Term,[quoted(true),numbervars(true)]) .
```

The output produced by `writeq/1/2` is generally suitable for re-input (e.g. with `read_term/2/3`), provided the same operator definitions are used for input as at the time of the output, and provided the special structure `'$VAR'(...)` (see `numbervars` under `write_term/2/3`) was not used.

Example

```
[user] ?- writeq(hello),nl.<ENTER>
hello

yes

[user] ?- writeq('new\nline'),nl.<ENTER>
'new\nline'

yes

[user] ?- writeq('Robert\'s daughter'),nl.<ENTER>
'Robert''s daughter'

yes

[user] ?- writeq('+'(3,'*'(7,8.3))),nl.<ENTER>
3 + 7 * 8.3

yes
```

Standard

These predicates comply with the definition of the ISO standard for Prolog.

See also

write/1/2, write_term/2/3, write_canonical/1/2, read_term/2/3

Convert a term into an atom

writeq_atom(@*Term*, ?*Atom* **)**

The predicate `writeq_atom/2` converts *Term* into an atom, taking the operator definitions into account, and unifies the result with *Atom*.

Conversion to an atom occurs as with `writeq/1`, i.e. special characters in atoms are handled according to Prolog syntax rules.

Arguments

Term	Term
Atom	Atom

Exceptions

type_error(atom)
The argument *Atom* must be a variable or an atom, but is a term of another type.

Hints

The call

```
writeq_atom(Term,Atom)
```

is identical to

```
open(string(test),write,S),
writeq(S,Term),
stream_control(S,string(Atom)),
close(S) .
```

An atom created by `writeq_atom/2` is generally suitable for re-input (e.g. with `parse_atom/6`), provided the same operator definitions are used for input as at the time of the output, and provided the special structure `'$VAR'(...)` (see `numbervars` under `write_term/2/3`) was not used.

Example

```
[user] ?- writeq_atom(a(b+c),A).<ENTER>

A       = 'a(b + c)' <ENTER>
```

```
yes

[user] ?- writeq_atom(['*b','\'',3.0],A).<ENTER>

A       = '['*b'',''''''''',3.0]' <ENTER>

yes
```

Standard

This predicate is not required by the ISO standard for Prolog. Therefore, it is not available in ISO compatibility mode (see Prolog flag `iso`).

See also

write_atom/2, writeq/1/2, write_term/2/3, parse_atom/6, stream_control/2

Chapter 5

Overview by functionality of debugger commands

This section contains an overview of the built-in debugger commands, arranged into groups according to their functionality.

The functor of each debugger command is printed in bold type. This table contains only the long forms of the commands.

A command that is only available in the alpha surface of the debugger is identified by the character #. Normally these commands are not available under the command surface of the debugger (stream). Under the Motif surface likewise these commands are not available, but it supplies graphic elements like buttons, menus and scrollbars.

Breakpoints

Command	Short	Functionality
activate_stop	**as**	Reactivate current breakpoints
activate_stop(Numbers)	**as**(...)	Reactivate breakpoints
deactivate_stop	**ds**	Deactivate all breakpoints temporarily
deactivate_stop(Numbers)	**ds**(...)	Deactivate breakpoint temporarily
remove_stop	**rs**	Remove all current explicit breakpoints
remove_stop(Numbers)	**rs**(...)	Remove explicit breakpoints
stop(Breakpoint, Ports, Conditions, Actions)	**st**(...)	Set breakpoint
stop(MonitoredVariable, BindingType, Conditions, Actions)	**st**(...)	Set breakpoint to the monitoring of variables
stop_interactive(Breakpoint)	**si**(...)	Set interactive breakpoint
stop_interactive(Breakpoint, Ports)	**si**(...)	Set interactive breakpoint

stop_interactive(Breakpoint, Ports, Conditions, Actions)	**si**(...)	Set interactive breakpoint
stop_interactive(MonitoredVariable, BindingType, Conditions, Actions)	**si**(...)	Set interactive breakpoint to the monitoring of variables
activate_stop	**as**	Reactivate current breakpoints
activate_stop(Numbers)	**as**(...)	Reactivate breakpoints
deactivate_stop	**ds**	Deactivate all breakpoints temporarily
deactivate_stop(Numbers)	**ds**(...)	Deactivate breakpoint temporarily
remove_stop	**rs**	Remove all current explicit breakpoints
remove_stop(Numbers)	**rs**(...)	Remove explicit breakpoints
stop(Breakpoint, Ports, Conditions, Actions)	**st**(...)	Set breakpoint
stop(MonitoredVariable, BindingType, Conditions, Actions)	**st**(...)	Set breakpoint to the monitoring of variables
stop_interactive(Breakpoint)	**si**(...)	Set interactive breakpoint
stop_interactive(Breakpoint, Ports)	**si**(...)	Set interactive breakpoint
stop_interactive(Breakpoint, Ports, Conditions, Actions)	**si**(...)	Set interactive breakpoint
stop_interactive(MonitoredVariable, BindingType, Conditions, Actions)	**si**(...)	Set interactive breakpoint to the monitoring of variables

Control commands

Command	**Short**	**Functionality**
abort	**A**	Terminate interactive debugger
break	**B**	Start new input loop
call(Goal) [@ +*Module*]	**ca**(...)	Execute a Prolog goal
halt	**H**	Terminate debugger and IF/Prolog
system	**S**	Interrupt interactive debugger and start the shell
abort	**A**	Terminate interactive debugger
break	**B**	Start new input loop
call(Goal) [@ +*Module*]	**ca**(...)	Execute a Prolog goal
halt	**H**	Terminate debugger and IF/Prolog

system	**S**	Interrupt interactive debugger and start the shell

Configuration

Command	Short	Functionality
port(*Ports*, *Switch*)	**p**(...)	Set debugger ports
set_debug_depth(Depth)	**sdd**(...)	Set structure depth
# **set_history**(Number)	**sh**(...)	Configure history
set_trace_depth(*Depth*)	**std**(...)	Set structure depth for trace outputs
# **set_trace_length**(Length)	**stl**(...)	Set length of trace field
trace(Switch)	**t**(...)	Switch trace mode
trace_port(Ports, Switch)	**tp**(...)	Set trace ports
port(*Ports*, *Switch*)	**p**(...)	Set debugger ports
set_debug_depth(Depth)	**sdd**(...)	Set structure depth
# **set_history**(Number)	**sh**(...)	Configure history
set_trace_depth(*Depth*)	**std**(...)	Set structure depth for trace outputs
# **set_trace_length**(Length)	**stl**(...)	Set length of trace field
trace(Switch)	**t**(...)	Switch trace mode
trace_port(Ports, Switch)	**tp**(...)	Set trace ports

Data manipulation

Command	Short	Functionality
unify(Variable, Value)	...=...	Unify variable
unify(Variable, Value)	...=...	Unify variable

Execution control

Command	Short	Functionality
continue	**c**	Continue to next breakpoint
fast_skip	**fs**	Accelerate execution to exit of the subsequent subgoal
fast_skip(Number)	**fs**(...)	Accelerate execution to exit of a subsequent subgoal
nonstop	**no**	Continue without stopping

skip	**s**	Execute up to exit of the subgoal
skip(Number)	**s**(...)	Execute up to exit of a subsequent subgoal
step	↩	Execute up to next monitored port
continue	**c**	Continue to next breakpoint
fast_skip	**fs**	Accelerate execution to exit of the subsequent subgoal
fast_skip(Number)	**fs**(...)	Accelerate execution to exit of a subsequent subgoal
nonstop	**no**	Continue without stopping
skip	**s**	Execute up to exit of the subgoal
skip(Number)	**s**(...)	Execute up to exit of a subsequent subgoal
step	↩	Execute up to next monitored port

Display information

Command	**Short**	**Functionality**
help	**h**	Display help information
# **snapshot**	**sn**	Store or append field in previous used file
# **snapshot**(File)	**sn**(...)	Store field in file
# **snapshot**(File, Mode)	**sn**(...)	Store or append field in file
trace	**t**	Logging passes through ports
# **view_and_box**	**&**	Display the goal box in the window
# **view_or_box**	**#**	Display the clause box in the window
# **view_history**	**vh**	Display the goal or clause box in the history list
# **view_trace**	**vt**	Display the trace field in the window
view_ancestors	**a**	Show dynamic call chain
view_ancestors(Number)	**a**(...)	Show dynamic call chain
view_configuration	**vco**	Display default values
view_constraints	**vc**	Display constraints
view_module	**vm**	Display list of predicates in module
view_module(Module)	**vm**(...)	Display list of predicates in module
view_predicate	**v**	Display listing of a predicate
view_predicate(Predicate)	**v**(...)	Display listing of a predicate
view_stops	**vs**	Display explicit breakpoints
view_variable	**vv**	Display variable constraints
view_variable(VariableName)	**vv**(...)	Display variable constraints
help	**h**	Display help information
# **snapshot**	**sn**	Store or append field in previous used file
# **snapshot**(File)	**sn**(...)	Store field in file

# **snapshot**(File, Mode)	**sn**(...)	Store or append field in file
trace	**t**	Logging passes through ports
# **view_and_box**	**&**	Display the goal box in the window
# **view_or_box**	**#**	Display the clause box in the window
# **view_history**	**vh**	Display the goal or clause box in the history list
# **view_trace**	**vt**	Display the trace field in the window
view_ancestors	**a**	Show dynamic call chain
view_ancestors(Number)	**a**(...)	Show dynamic call chain
view_configuration	**vco**	Display default values
view_constraints	**vc**	Display constraints
view_module	**vm**	Display list of predicates in module
view_module(Module)	**vm**(...)	Display list of predicates in module
view_predicate	**v**	Display listing of a predicate
view_predicate(Predicate)	**v**(...)	Display listing of a predicate
view_stops	**vs**	Display explicit breakpoints
view_variable	**vv**	Display variable constraints
view_variable(VariableName)	**vv**(...)	Display variable constraints

Jump commands

Command	Short	Functionality
back_clause	**bc**	Branch back to clause head
back_parent	**bp**	Branch back to CALL port of parent goal
back_subgoal	**b**	Branch back to CALL port of subgoal
error(Error, Info)	**err**(...)	Raise an exception
exit	**x**	Force current subgoal to succeed
fail	**f**	Force the current subgoal to fail
back_clause	**bc**	Branch back to clause head
back_parent	**bp**	Branch back to CALL port of parent goal
back_subgoal	**b**	Branch back to CALL port of subgoal
error(Error, Info)	**err**(...)	Raise an exception
exit	**x**	Force current subgoal to succeed
fail	**f**	Force the current subgoal to fail

Scrolling and paging

Command	Short	Functionality
# +		Scroll line by line forwards

#	-		Scroll line by line backwards
#	++		Scroll page by page forwards
#	--		Scroll page by page backwards
#	**forward**		Scroll line by line forwards
#	**backward**		Scroll line by line backwards
#	**fast_forward**		Scroll page by page forwards
#	**fast_backward**		Scroll page by page backwards
#	**view_previous_screen**	<	Display the previous goal or clause box
#	**view_last_screen**	<<	Display the last goal or clause box stored
#	**view_next_screen**	>	Display the next goal or clause box
#	**view_first_screen**	>>	Display the first goal or clause box stored
#	+		Scroll line by line forwards
#	-		Scroll line by line backwards
#	++		Scroll page by page forwards
#	--		Scroll page by page backwards
#	**forward**		Scroll line by line forwards
#	**backward**		Scroll line by line backwards
#	**fast_forward**		Scroll page by page forwards
#	**fast_backward**		Scroll page by page backwards
#	**view_previous_screen**	<	Display the previous goal or clause box
#	**view_last_screen**	<<	Display the last goal or clause box stored
#	**view_next_screen**	>	Display the next goal or clause box
#	**view_first_screen**	>>	Display the first goal or clause box stored

Chapter 6

Reference section on debugger commands

This section contains descriptions of the commands of the debugger in alphabetical order.

A detailed description of debugger and examples, can be found in the section on the *Debugger* in the *IF/Prolog User's Guide* [2]. Debugger commands which are adjacent to each other and which are similar in function are described together.

Debugger commands that are only available in the alpha interface of the debugger are specially marked.

Most commands have a short form which is also listed.

In debugger commands, which in addition to their arguments are supplied with information on the calling module, the argument list (as in the example) is followed by the text `[ @ +`*Module* `]`. In this case a module specification can be specified by `@/2` similar to meta predicates.

Scrolling

forward
+

backward
-

fast_forward
++

fast_backward
--

These commands enable the user to scroll in the current field. + and - scroll line by line forwards and backwards respectively. ++ and -- scroll page by page forwards and backwards respectively. If the user has already reached the field boundaries, these commands have no effect.

A plus (+) sign is displayed in the information area of the current field if the field extends beyond the lower edge of the window, and a minus (-) sign is displayed if the field extends beyond the upper edge of the window.

Exceptions

domain_error(arity)
The command `forward`, `backward`, `fast_forward` or `fast_backward` was called with an incorrect number of parameters.

Hints

These commands are only available in the alpha interface of the debugger.

Terminate interactive debugger

abort **A**

The `abort/0` command aborts the debugger session. The predicate being tested is evaluated with *fail*.

Exceptions

domain_error(arity)
: The command `abort/0` was called with an incorrect number of parameters.

See also

break/0, call/1

Reactivate breakpoint

activate_stop
as

activate_stop(*Numbers* **)**
as(*Numbers* **)**

The `activate_stop/1` command activates the explicit breakpoints specified using the *Numbers* argument. If *Numbers* is omitted, all current explicit breakpoints are activated.

An integer or list of integers may be specified as the argument.

If a list is specified, all the breakpoints included in the list are activated.

Arguments

Numbers	Integer or integer expression, or a list of integers or integer expressions, *Numbers* >= 0

Exceptions

instantiation_error
The argument *Numbers* or a subterm must not be a variable, but a variable was specified.

type_error(integer_or_list)
The argument *Numbers* must be an integer, an integer expression or a list of integers or integer expressions, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Numbers* must be a positive integer, but is negative.

existence_error(stop)
The argument *Numbers* does not specify an existing breakpoint.

domain_error(arity)
The command `activate_stop/0/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Numbers*, all the exceptions for `is/2` can occur.

Hints

Breakpoints can be deactivated with the `deactivate_stop/0/1` command.

See also

deactivate_stop/0/1, remove_stop/0/1, stop1/2/4, stop_interactive/1/2/4, view_stops/0

Branch back to clause head

back_clause
bc

The `back_clause/0` command is used to branch back to the head of the current clause (`TRYMATCH` port). If the command is entered at a clause head, the debugger branches back to the clause head of the ancestor clause. If no ancestor clause exists, the debugger reports an error.

The debugger stops at the goal of the `back_clause/0` command. Because execution of this command is equivalent to resetting the system to a previous state, variable instantiations may be undone.

Exceptions

existence_error(clause)
: The command `back_clause/0` cannot be executed at this point in the proof tree because the current predicate does not possess any clauses.

existence_error(previous_clause)
: The command `back_clause/0` cannot be executed at this point in the proof tree because the current breakpoint is already located at the first clause of the current predicate.

domain_error(arity)
: The command `back_clause/0` was called with an incorrect number of parameters.

See also

back_subgoal/0, back_parent/0

Branch back to CALL port of parent goal

back_parent
bp

The `back_parent/0` is used to branch back to the `CALL` port of the parent goal in the proof tree and stop there. Because execution of this command is equivalent to resetting the system to a previous state, variable instantiations may be undone. If no parent goal exists, the debugger reports an error.

Exceptions

existence_error(parent)
The command `back_parent/0` cannot be executed at this point in the proof tree.

domain_error(arity)
The command `back_parent/0` was called with an incorrect number of parameters.

See also

back_subgoal/0, back_clause/0

Branch back to CALL port of subgoal

back_subgoal
b

The `back_subgoal/0` command is used to branch back to the `CALL` or `WAKEUP` port of the current subgoal. If the command is entered at a `CALL` or `WAKEUP` port, the debugger branches back to the `CALL` or `WAKEUP` port of the ancestor goal.

The debugger stops at the goal of the `back_subgoal/0` command. Because execution of this command is equivalent to resetting the system to a previous state, variable instantiations may be undone. If no ancestor goal exists, the debugger reports an error.

Exceptions

existence_error(previous_subgoal)
The command `back_subgoal/0` cannot be executed at this point in the proof tree.

domain_error(arity)
The command `back_subgoal/0` was called with an incorrect number of parameters.

See also

back_parent/0, back_clause/0

Start new input loop

break
B

The `break/0` command starts a new input loop. The debugger is interrupted for the duration of the input loop.

Exceptions

domain_error(arity)
The command `break/0` was called with an incorrect number of parameters.

See also

abort/0, call/1

Execute a Prolog goal

call(*Goal* **)** `[` **@** `+`*Module* `]`
ca(*Goal* **)** `[` **@** `+`*Module* `]`

The `call/1` command is used to call the goal *Goal*. The failure or success of this operation has no effect on the remainder of the debugger session.

If a module argument is specified, *Goal* is called in the context of this module.

Arguments

Goal Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

domain_error(arity)
The command `call/1` was called with an incorrect number of parameters.

See also

abort/0, break/0, call/1, set_prolog_flag/2

Continue to next breakpoint

continue
c

The `continue/0` command resumes the debugging session, and the debugger stops at the next breakpoint. All implicit breakpoints except ERROR are deactivated.

Exceptions

domain_error(arity)
The command `continue/0` was called with an incorrect number of parameters.

See also

step/0, skip/0/1, nonstop/0, fast_skip/0/1

Deactivate breakpoint temporarily

deactivate_stop
ds

deactivate_stop(*Numbers* **)**
ds(*Numbers* **)**

The `deactivate_stop/1` command deactivates the explicit breakpoints specified using the *Numbers* argument.

If *Numbers* is omitted, all current explicit breakpoints are deactivated.

An integer or list of integers may be specified as the argument. If a list is specified, all the breakpoints included in the list are deactivated.

Arguments

Numbers	Integer or integer expression, or a list of integers or integer expressions.

Exceptions

instantiation_error
The argument *Numbers* or a subterm must not be a variable, but a variable was specified.

type_error(integer_or_list)
The argument *Numbers* must be an integer, an integer expression or a list of integers or integer expressions, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Numbers* must be a positive integer, but is negative.

existence_error(stop)
The argument *Numbers* does not specify an existing breakpoint.

domain_error(arity)
The command `deactivate_stop/0/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Numbers*, all the exceptions for `is/2` can occur.

Hints

Breakpoints can be reactivated with the `activate_stop/0/1` command.

See also

activate_stop/0/1, remove_stop/0/1, stop/1/2/4, stop_interactive/1/2/4, view_stops/0

Raise an exception

error(*Error*, *Info* **)**
err(*Error*, *Info* **)**

The `error/2` command is used to raise an exception with the arguments *Error* and *Info* in the current subgoal. This means that the debugger branches to the `ERROR` port in the current box.

Arguments

Error	Structure or atom
Info	List of elements like *InfoName* = *Wert*
InfoName	Atom
Wert	Term

Exceptions

instantiation_error
The argument *Error* or *Info* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *Info* must be a list, but is a term of another type.

type_error(assignment)
An element of *Info* is not an assignment with the format *InfoName* `=` *Wert*.

domain_error(arity)
The command `error/2` was called with an incorrect number of parameters.

See also

exit/0, fail/0, predicate exception/2

Force current subgoal to succeed

exit
x

The `exit/0` command is used to force the current subgoal to succeed. This may, however, result in no internal processing of the subgoal being performed. This, in turn, means that variable instantiations or side effects may not take place.

The debugger branches to the `EXIT` port in the current box and stops there.

Exceptions

domain_error(arity)
The command `exit/0` was called with an incorrect number of parameters.

See also

error/2, fail/0

Force the current subgoal to fail

fail **f**

The `fail/0` command forces the current subgoal to fail; the debugger branches to the `FAIL`, `FAILMATCH` or `FAILBODY` port in the current box and stops there.

Exceptions

domain_error(arity)
: The command `fail/0` was called with an incorrect number of parameters.

See also

error/2, exit/0

Accelerate execution to exit of a subsequent subgoal

fast_skip
fs

fast_skip(*Number* **)**
fs(*Number* **)**

The `fast_skip/0/1` command resumes the debugger session; the debugger stops at the exit (`EXIT`) of the (current+*Number*)th subgoal (starting at the current subgoal) and switches to interactive mode.

If *Number* is larger than the number of the remaining subgoals in the current clause, the debugger is positioned after the last subgoal of the current clause, it stops at the next subgoal of the parent clause. If the execution of the (current+*Number*)th subgoal fails, the debugger stops at the exit `FAIL/ERROR` of the subgoal involved.

The debugger does not metainterpret the subgoals; they are called directly instead. This means that breakpoints in subgoals are ignored.

The command `fast_skip/0` is equivalent to the command `fast_skip`(0).

Arguments

Number	Integer or integer expression, *Number* >= 0.

Exceptions

instantiation_error
The argument *Number* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Number* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Number* must be a positive integer, but is negative.

domain_error(arity)
The command `fast_skip/0/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

See also

step/0, continue/0, nonstop/0, skip/0/1

Terminate debugger and IF/Prolog

halt
H

The `halt/0` command aborts the debugger and terminates IF/Prolog. The command is equivalent to `call(halt)`.

Exceptions

domain_error(arity)
The command `halt/0` was called with an incorrect number of parameters.

See also

call/1, break/0, abort/0, predicate halt/0

Display help information

help
h

This command displays a list of commands with a brief description of each as help information.

Exceptions

domain_error(arity)
The command `help/0` was called with an incorrect number of parameters.

Hints

The information described here is formatted and supplemented in accordance with the interface. The appearance of the output therefore depends on the interface.

Continue without stopping

nonstop
no

The `nonstop/0` command is resumes the debugging session. The debugger does not stop throughout subsequent execution until it reaches the last port before it exits. It then switches to interactive mode.

Exceptions

domain_error(arity)
The command `nonstop/0` was called with an incorrect number of parameters.

See also

step/0, skip/0/1, continue/0, fast_skip/0/1

Set debugger port

port(*Ports*, *Switch* **)**
p(*Ports*, *Switch* **)**

The `port/2` command can be used to set the debugger ports. One of the following atoms, or a list of the following atoms, may be specified as the port(s).

Port	Abbreviation	Meaning
call	c	`CALL` port
enterbody	eb	`ENTERBODY` port
fail	f	`FAIL` port
failbody	fb	`FAILBODY` port
failmatch	fm	`FAILMATCH` port
redo	r	`REDO` port
redobody	rb	`REDOBODY` port
suspend	s	`SUSPEND` port
trymatch	tm	`TRYMATCH` port
wakeup	w	`WAKEUP` port
exit	x	`EXIT` port
exitbody	xb	`EXITBODY` port
error	er	`ERROR` port

The following designations may be used for frequently used combinations of ports:

Designation	Meaning
all	all ports
inner	enterbody, failbody, failmatch, redobody, trymatch,exitbody
outer	call, fail, redo, wakeup, suspend, exit, error

Arguments

Ports	Atom or list: call | enterbody | fail | failbody | failmatch | redo | redobody | suspend | trymatch | wakeup | exit | exitbody | error | all | inner | outer | c | eb | f | fb | fm | r | rb | s | tm | w | x | xb | er
Switch	Atom: on | off

Exceptions

instantiation_error
The argument *Ports* or *Switch* or a subterm must not be a variable, but a variable was specified.

type_error(atom_or_list)
The argument *Ports* must be an atom or a list of atoms, but is a term of another type.

domain_error(port)
The argument *Ports* is not valid for a port of the box model.

type_error(atom)
The argument *Switch* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Switch* must be one of the atoms `on` or `off`, but is another atom.

domain_error(arity)
The command `port/2` was called with an incorrect number of parameters.

See also

trace_port/2

Remove explicit breakpoints

remove_stop
rs

remove_stop(*Numbers* **)**
rs(*Numbers* **)**

The `remove_stop/1` command removes the breakpoints specified using the *Numbers* argument.

If *Numbers* is omitted, all current breakpoints are removed.

An integer or list of integers may be specified as the argument. If a list is specified, all the breakpoints included in the list are removed.

If the breakpoint or one of the breakpoints does not exist, an error message is generated. In this case, none of the specified breakpoints are removed.

Arguments

Numbers	Integer or integer expression, or a list of integers or integer expressions, *Numbers* >= 0

Exceptions

instantiation_error
The argument *Numbers* or a subterm must not be a variable, but a variable was specified.

type_error(integer_or_list)
The argument *Numbers* must be an integer, an integer expression or a list of integers or integer expressions, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Numbers* must be a positive integer, but is negative.

existence_error(stop)
The argument *Numbers* does not specify an existing breakpoint.

domain_error(arity)
The command `remove_stop/0/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Numbers*, all the exceptions for `is/2` can occur.

See also

stop/1/2/4, stop_interactive/1/2/4, view_stops/0, activate_stop/0/1, deactivate_stop/0/1

Set structure depth

set_debug_depth(*Depth* **)**
sdd(*Depth* **)**

The `set_debug_depth/1` command is used to specify the structure depth to be used for displaying terms.

Atoms have a structure depth of 0. If terms have a greater nesting depth, the terms of the last display depth are displayed with the format *Functor(*)* (or '...' in lists).

There is no restriction if 0 is specified.

Arguments

Depth	Integer or integer expression, *Depth* >= 0

Exceptions

instantiation_error
The argument *Depth* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Depth* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Depth* must be a positive integer, but is negative.

domain_error(arity)
The command `set_debug_depth/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Depth*, all the exceptions for `is/2` can occur.

See also

set_trace_depth/1

Configure history

set_history(*Number***)**
sh(*Number***)**

The `set_history/1` command is used to specify the number of goal and clause boxes to be stored.

The goal and clause boxes at the interactive breakpoints are stored in chronological order.

Arguments

Number	Integer or integer expression, number of boxes to be stored.

Exceptions

instantiation_error
The argument *Number* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Number* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Number* must be a positive integer, but is negative.

domain_error(arity)
The command `set_history` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

Hints

This command is only available in the alpha interface of the debugger.

See also

view_history/0, view_first_screen/0, view_last_screen/0, view_next_screen/0, view_previous_screen/0

Set structure depth for trace outputs

set_trace_depth(*Depth* **)**
std(*Depth* **)**

The `set_trace_depth/1` command is used to specify the structure depth for trace outputs to be used for displaying terms.

Atoms have a structure depth of 0. If terms have a greater nesting depth, the terms of the last display depth are displayed with the format *Functor(*)* (or '...' in lists).

There is no restriction if 0 is specified.

Arguments

Depth	Integer or integer expression, *Depth* >= 0

Exceptions

instantiation_error
The argument *Depth* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Depth* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Depth* must be a positive integer, but is negative.

domain_error(arity)
The command `set_trace_depth/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Depth*, all the exceptions for `is/2` can occur.

See also

set_debug_depth/1

Set length of trace field

set_trace_length(*Length* **)**
stl(*Length* **)**

The `set_trace_length/1` command is used to specify the length of the trace field.

The number of trace messages stored is at least the number specified. For reasons of runtime optimization, more trace messages than the number specified may be stored at times.

Arguments

Length	Integer or integer expression, *Length* >= 0

Exceptions

instantiation_error
The argument *Length* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Length* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Length* must be a positive integer, but is negative.

domain_error(arity)
The command `set_trace_length/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Length*, all the exceptions for `is/2` can occur.

Hints

This command is only available in the alpha interface of the debugger.

See also

view_trace/0

Execute up to exit of a subsequent subgoal

skip
s

skip(*Number* **)**
s(*Number* **)**

The `skip/0/1` command is resumes the debugger session; the debugger stops at the exit (`EXIT`) of the (current+*Number*)th subgoal (starting at the current subgoal) and switches to interactive mode.

If *Number* is larger than the number of the remaining subgoals in the current clause, the debugger is positioned after the last subgoal of the current clause, it stops at the next subgoal of the parent clause. If the (current+*Number*)th subgoal fails, the debugger stops at the exit `FAIL/ERROR` of the subgoal involved.

The `skip/0` command is equivalent to `skip`(0).

Arguments

Number	Integer or integer expression, *Number* >= 0

Exceptions

instantiation_error
The argument *Number* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Number* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Number* must be a positive integer, but is negative.

domain_error(arity)
The command `skip/0/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

See also

step/0, continue/0, nonstop/0, fast_skip/0/1

Store field

snapshot
sn

snapshot(*File***)**
sn(*File***)**

snapshot(*File***,** *Mode***)**
sn(*File***,** *Mode***)**

The `snapshot/0/1/2` command writes the current field to a file. The entire field in its full length and width is written to the file, taking valid structure depth settings into account.

If *Mode* is `write`, the previous contents of the file are deleted. If *Mode* is `append`, the field is appended to the file.

The command `snapshot/1` is equivalent to `snapshot/2` with *Mode* `write`.

The command `snapshot/0` is equivalent to `snapshot/2` with *Mode* `append` where *File* refers to the last file opened with `snapshot/1/2`.

Arguments

File	Atom
Mode	Atom: write | append

Exceptions

instantiation_error
The argument *File or Mode* must not be a variable, but a variable was specified.

type_error(atom)
The argument *File or Mode* must be an atom, but is a term of another type.

domain_error(open_mode)
The value of the argument *Mode* is invalid for the mode of a stream.

permission_error(open,source_sink)
The argument *File* specifies a file (or a different I/O medium) that cannot be opened.

system_error(write)
system_error(close)
system_error(flush_output)
system_error(interrupted)
The indicated error was reported by the operating system when writing on the output stream *File*.

existence_error(snapshot_file)
The command `snapshot/0` was used although no snapshot file exists yet.

domain_error(arity)
The command `snapshot/0/1/2` was called with an incorrect number of parameters.

Hints

These commands are only available in the alpha interface of the debugger.

Execute up to next monitored port

step
↩

The `step/0` command is resumes the debugger session; the debugger continues to the next breakpoint and switches to interactive mode, if necessary. All implicit breakpoints are then activated.

This command can also be used to trace processing step by step, i.e. from port to port (for which `on` has been set).

If this command is used in the action list of a breakpoint, it must be written as *step*. The interpretation of ↩ as the abbreviation of *step* depends on the interface.

Exceptions

domain_error(arity)
The command `step/0` was called with an incorrect number of parameters.

Hints

For the stream interface, an empty input (↩) is synonymous with *step*.

In the case of the alpha interface, empty input (↩) will either be used to switch between the various display fields on the interface or will be considered synonymous with `step`, depending of the context in which it is used.

See also

skip/0/1, continue/0, nonstop/0, fast_skip/0/1

Set explicit breakpoint

stop(*Breakpoint*, *Ports*, *Conditions*, *Actions* **)**
stop(*MonitoredVariable*, *BindingType*, *Conditions*, *Actions* **)**
st(*Breakpoint*, *Ports*, *Conditions*, *Actions* **)**
st(*MonitoredVariable*, *BindingType*, *Conditions*, *Actions* **)**

stop_interactive(*Breakpoint* **)**
si(*Breakpoint* **)**

stop_interactive(*Breakpoint*, *Ports* **)**
si(*Breakpoint*, *Ports* **)**

stop_interactive(*Breakpoint*, *Ports*, *Conditions*, *Actions* **)**
stop_interactive(*MonitoredVariable*, *BindingType*, *Conditions*, *Actions* **)**
si(*Breakpoint*, *Ports*, *Conditions*, *Actions* **)**
si(*MonitoredVariable*, *BindingType*, *Conditions*, *Actions* **)**

The commands `stop/4` and `stop_interactive/1/2/4` are used to declare explicit breakpoints. The `stop/4` command defines a breakpoint and the `stop_interactive/1/2/4` command defines an interactive breakpoint. There are two kinds of breakpoints:

Event-driven breakpoints

An event-driven breakpoint is defined by specifying *MonitoredVariable* and *BindingType*. *MonitoredVariable* indicates the variable in the clause *ClauseNumber* to be monitored.

The atom `bind` or `modify` may be specified as the *BindingType*. `bind` refers to the monitoring of the binding of the variable *MonitoredVariable* to a term. The binding type `modify` includes the functionality of `bind` but automatically extends the monitoring to include any variables in the bound term.

Explicit breakpoints

The *Breakpoint* is defined by specifying a predicate indicator, or a predicate indicator and clause number, or a predicate indicator, clause number and subgoal number.

The debugger checks whether the predicate has been defined when the breakpoint is set.

The ports at which the debugger is to stop in the relevant box is specified in the *Ports* argument. One of the following atoms, or a list of the following atoms, may be specified as the port:

Port	Abbreviation	Meaning
call	c	`CALL` port
enterbody	eb	`ENTERBODY` port
fail	f	`FAIL` port
failbody	fb	`FAILBODY` port
failmatch	fm	`FAILMATCH` port
redo	r	`REDO` port
redobody	rb	`REDOBODY` port
suspend	s	`SUSPEND` port
trymatch	tm	`TRYMATCH` port
wakeup	w	`WAKEUP` port
exit	x	`EXIT` port
exitbody	xb	`EXITBODY` port
error	er	`ERROR` port

The following designations may be used for frequently used combinations of ports:

Designation	Meaning
all	all ports
inner	enterbody, failbody, failmatch, redobody, trymatch,exitbody
outer	call, fail, redo, wakeup, suspend, exit, error

If `stop_interactive`(*Breakpoint*) is used, the breakpoint is set at the call port.

Conditions enables the user to restrict further the criterion that determines whether the debugger is to stop at a breakpoint. These conditions can be used to query a variable instantiation; the instance, tree depth and recursion depth can be used directly in integer expressions with the format *Keyword Operator Value*. The keyword *frequency* is the number of passes through the breakpoint, *depth* is the current search depth in the search tree, and *recursion*is the recursion depth in the current search tree.

If the condition *match(Head)* is specified, the debugger checks whether *Head* can be unified with the current subgoal. If the condition *(match(Head) :- Body)* is specified, the debugger also attempts to prove *Body*. If the unification test for *Head* or proof of *Body* is not successful, the breakpoint is ignored. Unification with the clause head does not have any effect on the goal proving operation, i.e. it cannot be used to instantiate variables in the goal being proved.

In the condition *(match(Head)*, *Head* refers to the specified subgoal if the *Breakpoint* is defined as *PredicateIndicator*, *ClauseNumber*, *SubGoalNumber*. The debugger checks whether *Body* can, in principle, be unified with the specified subgoal in *Breakpoint*. If this is not the case, an error message (`domain_error`) is displayed, and the breakpoint is not set.

One or more debugger commands of the command interface can be specified in *Actions*. These commands are executed in sequence at the breakpoint.

Arguments

Breakpoint	Structure or list with the format: •*PredicateIndicator* •[*PredicateIndicator*, *ClauseNumber*] •[*PredicateIndicator*, *ClauseNumber*, *SubGoalNumber*]
MonitoredVariable	List of the form: [*PredicateIndicator*, *ClauseNumber*, *Variable*]
PredicateIndicator	Structure with the format: •*Functor / Arity* •*Module: Functor / Arity*
ClauseNumber	Integer, *ClauseNumber* > 0
SubGoalNumber	Integer, *SubGoalNumber* > 0
Ports	Atom or list of atoms: call | enterbody | fail | failbody | failmatch | redo | redobody | suspend | trymatch | wakeup | exit | exitbody | error | all | inner | outer | c | eb | f | fb | fm | r | rb | s | tm | w | x | xb | er
Variable	Variable
BindingType	Atom: bind | modify
Conditions	Structure with the following format or list with the following optional elements: •*frequency* [=\\= | < | > | =< | >= | == | *mod*] Number •*depth* [=\\= | < | > | =< | >= | == | *mod*] Number •*recursion* [=\\= | < | > | =< | >= | == | *mod*] Number •*match(Head)* •*(match(Head) :- Body)*
Number	Integer or integer expression, *Number* >= 0.
Head	Term
Body	Term
Actions	Atom, structure or list of atoms and/or structures. (Debugger commands)

Exceptions

instantiation_error
The argument or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
The argument *PredicateIndicator* must be a predicate indicator with the above-mentioned format, but is a term of another type.

type_error(atom)
The name of the predicate *PredicateIndicator* or the specified *Module* must be an atom.

type_error(integer)
The arity of the predicate *PredicateIndicator* must be an integer.

existence_error(module)
The *Module* specified for the predicate *PredicateIndicator* does not exist.

domain_error(not_less_then_zero)
The arity of the predicate *PredicateIndicator* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *PredicateIndicator* must be an integer in the range 0..127, but is larger.

existence_error(procedure)
The predicate indicator *PredicateIndicator* does not specify an existing predicate.

permission_error(access_clause,static_procedure)
The predicate indicator *PredicateIndicator* specifies a predicate whose clauses cannot be accessed.

permission_error(access,system_module)
The predicate indicator *PredicateIndicator* specifies a predicate whose defining module is a system module.

type_error(integer)
The argument *ClauseNumber* must be an integer, but is a term of another type.

domain_error(greater_than_zero)
The argument *ClauseNumber* must be an integer greater than zero, but is negative or equal to zero.

existence_error(clause_number)
The argument *ClauseNumber* does not specify an existing clause.

type_error(atom_or_list)
The argument *Ports* must be an atom or a list of atoms, but is a term of another type.

domain_error(port)
The argument *Ports* is not valid for a port of the box model.

type_error(atom)
The argument *BindingType* must be an atom, but is a term of another type.

domain_error(binding)
The value of the argument *BindingType* is invalid for the binding of a variable.

domain_error(condition)
The value of the argument *Conditions* is an invalid condition for a breakpoint.

domain_error(match)
The argument *Conditions* contains a `match` condition that is not compatible with the specified predicate.

existence_error(command)
The argument *Actions* contains a command that is unknown to the debugger.

permission_error(apply,command)
The argument *Actions* contains a command that is not permitted at a breakpoint.

type_error(integer_or_variable)
The argument *SubGoalNumber/Variable* must be an integer or a variable, but is a term of another type.

domain_error(arity)
The command `stop/4` or `stop_interactive/1/2/4` was called with an incorrect number of parameters.

Hints

No breakpoint can be set at language constructs (such as `->`).

The breakpoints that are set are assigned unique numbers that can be used in the commands `remove_stop/1`, `activate_stop/1` and `deactivate_stop/1`. The numbers can be displayed by means of the `view_stops/0` command.

See also

remove_stop/0/1, activate_stop/0/1, deactivate_stop/0/1, view_stops/0

Interrupt interactive debugger and start the shell

system
S

The `system/0` command interrupts the debugger session and starts a shell.

The command is equivalent to `call(system)`. The debugger session is resumed once the shell is terminated.

Exceptions

domain_error(arity)
The command `system/0` was called with an incorrect number of parameters.

See also

call/1, predicate system/0

Logging passes through ports

trace
t

trace(*Switch* **)**
t(*Switch* **)**

The `trace/0` is used to log passes through the current port.

If the argument value `on` is specified for *Switch* in the `trace/1` command, all passes through ports for which `on` has been specified for tracing are to be logged.

If `off` is specified, trace logging is deactivated.

Arguments

Switch	Atom: on | off

Exceptions

instantiation_error
The argument *Switch* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Switch* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Switch* must be one of the atoms `on` or `off`, but is another atom.

domain_error(arity)
The command `trace/0/1` was called with an incorrect number of parameters.

Hints

The trace output is formatted in accordance with the interface. Therefore the output of the command depends on the interface.

Set trace ports

trace_port(*Ports*, *Switch* **)**
tp(*Ports*, *Switch* **)**

The `trace_port/2` is used to set the trace ports. One of the following atoms, or a list of atoms, may be specified as the port(s):

Port	Abbreviation	Meaning
call	c	`CALL` port
enterbody	eb	`ENTERBODY` port
fail	f	`FAIL` port
failbody	fb	`FAILBODY` port
failmatch	fm	`FAILMATCH` port
redo	r	`REDO` port
redobody	rb	`REDOBODY` port
suspend	s	`SUSPEND` port
trymatch	tm	`TRYMATCH` port
wakeup	w	`WAKEUP` port
exit	x	`EXIT` port
exitbody	xb	`EXITBODY` port
error	er	`ERROR` port

The following designations may be used for frequently used combinations of ports:

Designation	Meaning
all	all ports
inner	enterbody, failbody, failmatch, redobody, trymatch, exitbody
outer	call, fail, redo, wakeup, suspend, exit, error

Arguments

Ports	Atom or list of atoms: call | enterbody | fail | failbody | failmatch | redo | redobody | suspend | trymatch | wakeup | exit | exitbody | error | all | inner | outer | c | eb | f | fb | fm | r | rb | s | tm | w | x | xb | er
Switch	Atom: on | off

Exceptions

instantiation_error
The argument *Ports* or *Switch* or a subterm must not be a variable, but a variable was specified.

type_error(atom_or_list)
The argument *Ports* must be an atom or a list of atoms, but is a term of another type.

domain_error(port)
The argument *Ports* is not valid for a port of the box model.

type_error(atom)
The argument *Switch* must be an atom, but is a term of another type.

domain_error(on_or_off)
The argument *Switch* must be one of the atoms `on` or `off`, but is another atom.

domain_error(arity)
The command `trace_port/2` was called with an incorrect number of parameters.

See also

port/2

Unify variable

unify(*Variable*, *Value* **)**
Variable = *Value*

The `unify/2` command can be used to unify a variable in the current clause with a value.

Arguments

Variable	Variable
Value	Term

Exceptions

type_error(variable)
The argument *Variable* must be a variable, but is a term of another type.

domain_error(arity)
The command `unify/2` was called with an incorrect number of parameters.

Select display fields

view_and_box
&

view_or_box
#

view_history
vh

view_trace
vt

These commands are used to make the field involved the current field. The commands have the following meanings:

Command	Short form	Meaning
`view_and_box`	&	Display the goal box in the window
`view_or_box`	#	Display the clause box in the window
`view_history`	vh	Display the goal or clause box in the history list
`view_trace`	vt	Display the trace field in the window

Exceptions

domain_error(arity)
The command One of the commands was called with an incorrect number of parameters.

Hints

These commands are only available in the alpha interface of the debugger.

Some commands of the alpha-interface also effect a change of the display field.

See also

set_history/1, forward, backward, fast_forward, fast_backward, +/0, -/0, ++/0, --/0, set_trace_length/1

Show dynamic call chain

view_ancestors
a

view_ancestors(*Number* **)**
a(*Number* **)**

The `view_ancestors/0/1` command enables the user to show the current dynamic call chain.

The output can be restricted to the last *Number* ancestor(s) with the *Number* argument.

Arguments

Number	Integer or integer expression, *Number* >= 0

Exceptions

instantiation_error
The argument *Number* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
The argument *Number* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

domain_error(not_less_than_zero)
The argument *Number* must be a positive integer, but is negative.

domain_error(arity)
The command `view_ancestor/0/1` was called with an incorrect number of parameters.

Furthermore, if an arithmetic expression is specified for the argument *Number*, all the exceptions for `is/2` can occur.

Hints

The information described here is formatted in accordance with the interface. The appearance of the output therefore depends on the interface.

Display default values

view_configuration
vco

The `view_configuration/0` command displays the current settings (ports, etc.).

Exceptions

domain_error(arity)
The command `view_configuration/0` was called with an incorrect number of parameters.

Hints

The information described here is formatted in accordance with the interface. The appearance of the output therefore depends on the interface.

Display constraints

view_constraints
vc

The `view_constraints/0` commands displays a pending constraint subgoal in the context of its definition.

The command is only permitted if a constraint subgoal is pending.

Exceptions

existence_error(pending_constraints)
: The command `view_constraint/0` cannot be executed since there are no pending constraints.

domain_error(arity)
: The command `view_constraint/0` was called with an incorrect number of parameters.

Hints

The information described here is formatted in accordance with the interface. The appearance of the output therefore depends on the interface.

If the alpha interface is used, the message `CONSTRAINTS PENDING` is displayed in the information area if constraint subgoals are pending.

See also

view_variable/1

Display list of predicates in module

view_module
vm

view_module(*Module* **)**
vm(*Module* **)**

The `view_module/0/1` commands displays the predicates of *Module* from the Prolog database.

The predicates are displayed with the clause and subgoal numbers. These numbers can be used with `stop/4` and `stop_interactive/1/2/4` when a breakpoint is created.

If this command is called without parameters, the predicates from the current module are displayed.

Arguments

Module	Atom, name of a module

Exceptions

instantiation_error
The argument *Module* must not be a variable, but a variable was specified.

type_error(atom)
The argument *Module* must be an atom, but is a term of another type.

existence_error(module)
The argument *Module* is not the name of an existing module.

permission_error(access,system_module)
The predicate *Module* must not be executed for a system module.

domain_error(arity)
The command `view_module/0/1` was called with an incorrect number of parameters.

Hints

The information described here is formatted in accordance with the interface. The appearance of the output therefore depends on the interface.

See also

view_predicate/0/1, stop/4, stop_interactive/1/2/4

Display listing of a predicate

view_predicate
v

view_predicate(*Predicate* **)**
v(*Predicate* **)**

The `view_predicate/0/1` command displays the predicate *Predicate* from the Prolog database.

The predicates are displayed with the clause and subgoal numbers. These numbers can be used with `stop/4` and `stop_interactive/1/2/4` when a breakpoint is created.

If this command is called without parameters, the current predicate is displayed.

Arguments

Predicate	Expression with the format: •*Module*:*Functor*/*Arity* •*Functor* •*Module*:*Functor*
Functor	Atom
Module	Atom, name of a module
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Predicate* or a subterm must not be a variable, but a variable was specified.

type_error(predicate_indicator)
The argument *Predicate* must be a predicate indicator with the above-mentioned format, but is a term of another type.

type_error(atom)
The name of the predicate *Predicate* or the specified *Module* must be an atom.

type_error(integer)
The arity of the predicate *Predicate* must be an integer.

existence_error(module)
The *Module* specified for the predicate *Predicate* does not exist.

domain_error(not_less_then_zero)
The arity of the predicate *Predicate* must be a positive integer, but is negative.

representation_error(max_arity)
The arity of the predicate *Predicate* must be an integer in the range 0..127, but is larger.

existence_error(procedure)
The predicate indicator *Predicate* does not specify an existing predicate.

permission_error(access_clause,static_procedure)
The predicate indicator *Predicate* specifies a predicate whose clauses cannot be accessed.

permission_error(access,system_module)
The predicate indicator *Predicate* specifies a predicate whose defining module is a system module.

existence_error(current_predicate)
The command `view_predicate/0` cannot be executed since no *current* predicate exists.

domain_error(arity)
The command `view_predicate/0/1` was called with an incorrect number of parameters.

Hints

The information described here is formatted in accordance with the interface. The appearance of the output therefore depends on the interface.

See also

view_module/0/1

Display history

view_previous_screen
<

view_last_screen
<<

view_next_screen
>

view_first_screen
>>

These commands are used to make the field involved the history field. The commands have the following meanings:

Command	Short form	Meaning
`view_previous_screen`	`<`	Display the previous goal or clause box
`view_last_screen`	`<<`	Display the last goal or clause box stored
`view_next_screen`	`>`	Display the next goal or clause box
`view_first_screen`	`>>`	Display the first goal or clause box stored

Exceptions

existence_error(screen)
: The command `view_first_screen/0`, `view_last_screen/0`, `view_previous_screen/0` or `view_next_screen/0` cannot be executed since no appropriate screen dump exists.

domain_error(arity)
: The command `view_first_screen/0`, `view_last_screen/0`, `view_previous_screen/0` or `view_next_screen/0` was called with an incorrect number of parameters.

Hints

These commands are only available in the alpha interface of the debugger.

See also

view_history/0, set_history/1

Display explicit breakpoints

view_stops
vs

The `view_stops/0` command displays all explicit breakpoints with all additional information (activated or deactivated, etc.).

Each breakpoint has a number which can be used to remove, activate or deactivate individual breakpoints.

Exceptions

domain_error(arity)
The command `view_stops/0` was called with an incorrect number of parameters.

Hints

The information described here is formatted in accordance with the interface. The appearance of the output therefore depends on the interface.

See also

stop/4, stop_interactive/1/2/4, remove_stop/0/1, activate_stop/0/1, deactivate_stop/0/1

Display variable constraints

view_variable
vv

view_variable(*VariableName* **)**
vv(*VariableName* **)**

The `view_variable/1` command displays any constraints at the current breakpoint for the variable *VariableName*. *VariableName* must occur in the current clause. The currently valid range of values is also displayed for variables that have `const_domain` class constraints.

The `view_variable/0` command can be used to display all suspended constraints at the current breakpoint. Variables that do not occur in the current clause are displayed using their internal name.

The constraint subgoals are stored internally while the debugger is active. Information on the built-in predicates used to declare constraints does not exist for constraints specified before the debugger was activated. In such a case, the constraints are displayed in an internal format.

In the class `const_linear` inequations are transformed:

```
Expr1 $>= $Expr2 ==> Expr1 + Slack $= Expr2, Slack $>= 0.

Expr1 $>  $Expr2 ==> Expr1 + StrictSlack $= Expr2, StrictSlack $> 0.
```

The so-called *slack variables* are displayed with internal numbers.

Arguments

VariableName	Variable

Exceptions

type_error(variable)
: The argument *VariableName* must be a variable, but is a term of another type.

existence_error(variable)
: The argument *VariableName* specifies a variable that does not occur in the specified clause of the predicate.

existence_error(constraint_variable)
: The argument *VariableName* is not a constraint variable.

domain_error(arity)
The command `view_variable/0/1` was called with an incorrect number of parameters.

Hints

The information described here is formatted in accordance with the interface. The appearance of the output therefore depends on the interface.

See also

view_constraints/0

Chapter 7

Overview by functionality of C interface

This section contains an overview of the built in functions of the active and passive C interface arranged into groups according to their functionality.

The name of each C function is printed in bold type. Each argument is preceded by its data type. The result type of the function stands before the function name.

Active Prolog goals

C Function	Functionality
void **PrologClose**(t_cursor ActGoal)	Deactivate the current Prolog goal
BOOLEAN **PrologError**(t_cursor ActGoal, TERM *Term)	Query error
BOOLEAN **PrologFetch**(t_cursor ActGoal)	Find solutions for a goal
t_cursor **PrologOpen**(MODULE Module, TERM Goal, TERM VarList)	Create current Prolog goal

Analyze term

C Function	Functionality
TERM **TermArg**(ARITY Argno, TERM Term)	Access structure arguments

TERMTYPE **TermDecompose**(TERM Term, TERMINFO *Info)	Classify term
TERMTYPE **TermType**(TERM Term)	Classify term

Test Prolog terms

C Function	**Functionality**
BOOLEAN **TermIsAtom**(TERM Term, STRING *Name)	Test for atom
BOOLEAN **TermIsCompound**(TERM Term, STRING Functor, ARITY Arity, ...)	Test for structure
BOOLEAN **TermIsFloat**(TERM Term, double *Value)	Test for floating-point number
BOOLEAN **TermIsFloatExpression**(TERM Term, double *Value)	Test for floating-point expression
BOOLEAN **TermIsFunctor**(TERM Term, STRING Functor, ARITY Arity)	Test for structure
BOOLEAN **TermIsInteger**(TERM Term, long *Value)	Test for integer
BOOLEAN **TermIsIntegerExpression**(TERM Term, long *Value)	Test for integer expression
BOOLEAN **TermIsList**(TERM Term, TERM *Head, TERM *Tail)	Test for list
BOOLEAN **TermIsNil**(TERM Term)	Test for empty list
BOOLEAN **TermIsUniversal**(TERM Term, STRING Functor, ARITY Arity, TERM Args[])	Test for structure
BOOLEAN **TermIsVar**(TERM Term)	Test for variable

Test results (passive C interface)

C Function	Functionality
TERM **PrologGoal**(t_cursor ActGoal)	Prolog goal as term
BOOLEAN **PrologIsFloat**(t_cursor ActGoal, STRING VarName, double *Value)	Get floating-point number from variable
BOOLEAN **PrologIsInteger**(t_cursor ActGoal, STRING VarName, long *Value)	Get integer from variable
BOOLEAN **PrologIsString**(t_cursor ActGoal, STRING VarName, STRING *Name)	Get character string from variable
BOOLEAN **PrologIsTerm**(t_cursor ActGoal, STRING VarName, TERM *Term)	Get Prolog term from variable

Connect to Prolog

C Function	Functionality
void **Cboot**(void)	Link a C function to IF/Prolog
void **CPRED**(STRING Functor, ARITY Arity, CPREDFUN Function, size_t Size)	Add a complex C function to IF/Prolog
void **CPRIM**(STRING Functor, ARITY Arity, CPRIMFUN Function)	Add a simple C function to IF/Prolog
void **Cshutdown**(void)	Call a C function on IF/Prolog shutdown
void **MCPRED**(STRING Module, STRING Functor, ARITY Arity, CPREDFUN Function, size_t Size)	Add a complex C function to a module
void **MCPRIM**(STRING Module, STRING Functor, ARITY Arity, CPRIMFUN Function)	Add a simple C function to a module
TERM **PrologArg**(ARITY Argno)	Access predicate arguments

Control backtracking

C Function	Functionality
void **PrologEnableEpilog**(void)	Enable epilog
void **PrologUndo**(void)	Undo unification

Construct lists

C Function	Functionality
BOOLEAN **TermAddList**(TERM *Tail, TERM Term)	Add to a Prolog list
BOOLEAN **TermCloseList**(TERM Tail)	Close a Prolog list
TERM **TermMakeList**(TERM Head, TERM Tail)	Generate a Prolog list
TERM **TermOpenList**(TERM *Tail)	Generate a Prolog list

Construct Prolog terms

C Function	Functionality
TERM **TermMakeAtom**(STRING name)	Generate a Prolog atom
TERM **TermMakeCompound**(STRING Functor, ARITY Arity, ...)	Generate a Prolog structure
TERM **TermMakeFloat**(double Value)	Generate a Prolog floating-point number
TERM **TermMakeFunctor**(STRING functor, ARITY arity)	Generate a Prolog structure
TERM **TermMakeInteger**(long Value)	Generate a Prolog integer
TERM **TermMakeList**(TERM Head, TERM Tail)	Generate a Prolog list
TERM **TermMakeVar**(void)	Generate an empty list

TERM **TermMakeUniversal**(STRING Functor, ARITY Arity, TERM Args[])	Generate a Prolog structure
TERM **TermMakeVar**(void)	Generate a Prolog variable

Device driver function

C Function	Functionality
void **DeviceCreate**(t_device *Device)	Register device driver
BOOLEAN **StreamFlush**(t_stream Stream, long *Number)	Flush Prolog output buffer
t_iomode **StreamMode**(t_stream Stream)	Query stream mode

Formatted output

C Function	Functionality
size_t **fprintt**(FILE *FilePointer, STRING Format, ...)	Formatted output of Prolog terms and C data objects
size_t **outputlen**(STRING Format, ...)	Determine length of formatted output
size_t **printt**(STRING Format, ...)	Formatted output of Prolog terms and C data objects
size_t **sprintt**(char *Buffer, STRING Format, ...)	Formatted output of Prolog terms and C data objects

Initialize Prolog (passive C interface)

C Function	Functionality
BOOLEAN **EndProlog**(void)	Release memory space used by IF/Prolog

BOOLEAN **InitProlog**(int Number, STRING *ArgumentVector, FILE *Input, FILE *Output, FILE *Error)	Initialize IF/Prolog and define the standard media

Memory management

C Function	Functionality
void **TermCollect**(TERMCONTEXT context, ...)	Release term variables
void **TermCollectSequence**(TERM-CONTEXT context, size_t *N*, TERM **TermList*)	Release term variables
TERMCONTEXT **TermContext**(void)	Current set of term variables

Prepare Prolog terms

C Function	Functionality
STRING **ParseError**(int Number)	Assignment of syntax error number to syntax error message
BOOLEAN **ParseProlog**(STRING GoalAtom, t_parse_context *Context)	Parse a goal string and store the information
MODULE **PrologModule**(STRING Name)	Get Prolog module

Passive Prolog terms

C Function	Functionality
t_query **QueryCreate**(MODULE Module, TERM Goal, TERM VarList)	Create passive Prolog goal
void **QueryDispose**(t_query Query)	Release passive Prolog goal
t_cursor **QueryOpen**(t_query Query)	Activate a passive Prolog goal

Raise errors

C Function	Functionality
void **ErrorContextClear**(void)	Clear error context
BOOLEAN **ErrorContextGet**(ERRORCLASS *Errorclass, STRING *Parm1, STRING *Parm2, ARITY *Argno, TERM *Culprit)	Query error context
BOOLEAN **ErrorContextIsSet**(void)	Status of the error context
void **ErrorContextSet**(ERRORCLASS Errorclass, STRING Parm1, STRING Parm2, ARITY Argno, TERM Culprit)	Set error context

Unification

C Function	Functionality
BOOLEAN **TermUnify**(TERM Term1, TERM Term2)	Unify terms
BOOLEAN **TermUnifyAtom**(TERM Term, STRING Name)	Unify term with atom
BOOLEAN **TermUnifyCompound**(TERM Term, STRING Functor, ARITY Arity, ...)	Unify term with structure
BOOLEAN **TermUnifyFloat**(TERM Term, double Value)	Unify term with floating-point number
BOOLEAN **TermUnifyFunctor**(TERM Term, STRING Functor, ARITY Arity)	Unify term with structure
BOOLEAN **TermUnifyInteger**(TERM Term, long Value)	Unify term with integer
BOOLEAN **TermUnifyList**(TERM Term, TERM Head, TERM Tail)	Unify term with list
BOOLEAN **TermUnifyNil**(TERM Term)	Unify term with the empty list

BOOLEAN
TermUnifyUniversal(TERM Term,
STRING Functor, ARITY Arity,
TERM *Args[])
Unify term with structure

BOOLEAN
TermUnifyVar(TERM Term1,
TERM Term2)
Unify terms

Chapter 8

Reference section on C functions

This section contains descriptions of all functions of the C interface and the device driver interface in alphabetical order. A detailed description of all C-interfaces and more examples of the use of C-interfaces can be found in the *IF/Prolog User's Guide* [2].

The function prototype is noted in ANSI/ISO-C syntax notation.

Link a C function to IF/Prolog

```
#include "cpred.h"
```

```
void Cboot(void)
```

The function `Cboot()` is called automatically when IF/Prolog is started. It is used to link all the user defined C predicates and device drivers to IF/Prolog.

Hints

An empty `Cboot()` function supplied by he system is used if the user did not define an own `Cboot()` function.

See also

Cshutdown(), CPRED(), CPRIM(), MCPRIM(), MCPRED(), DeviceCreate()

Add a complex C function to IF/Prolog

```
#include "cpred.h"
```

```
void CPRED(STRING Functor, ARITY Arity, CPREDFUN Function,
size_t Size)
```

The function `CPRED()` is used to incorporate the predicate *Functor*/*Arity* into IF/Prolog. The predicate is implemented by the complex C function *Function*. A complex function is re-executable, i.e. allows backtracking.

Size specifies the size, in bytes, of the data structure needed for built-in backtracking.

The predicate is assigned to the module `user`.

Arguments

Functor	Character string
Arity	Integer, $0 \leq Arity \leq 127$
Function	C Function
Size	Integer

Hints

The function `CPRED()` should be called from the C function `Cboot()`, which is executed at startup of IF/Prolog.

See also

Cboot(), CPRIM(), MCPRIM(), MCPRED()

Add a simple C function to IF/Prolog

```
#include "cpred.h"
```

```
void CPRIM(STRING Functor, ARITY Arity, CPRIMFUN Function)
```

The function `CPRIM()` is used to incorporate the predicate *Functor*/*Arity* into IF/Prolog. The predicate is implemented by the simple C function *Function*. A simple function is deterministic, i.e. does not allow backtracking.

The predicate is assigned to the module `user`.

Arguments

Functor	Character string
Arity	Integer, $0 \leq Arity \leq 127$
Function	C Function

Hints

The function `CPRIM()` should be called from the C function `Cboot()`, which is executed at startup of IF/Prolog.

See also

Cboot(), CPRED(), MCPRIM(), MCPRED()

Call a C function on IF/Prolog shutdown

```
#include "cpred.h"
```

```
void Cshutdown(void)
```

The function `Cshutdown()` is called automatically on IF/Prolog shutdown (inside the `EndProlog()` function. It is used to call the user defined C code on IF/Prolog shutdown - e.g. releasing user allocated resources, ...

Hints

An empty `Cshutdown()` function supplied by he system is used if the user did not define an own `Cshutdown()` function. You can find it in the file `Cboot.c`.

See also

Cboot(), EndProlog()

Register device driver

```
#include "device.h"
```

```
void DeviceCreate(t_device *Device)
```

The function `DeviceCreate()` declares the new device driver to IF/Prolog.

The argument *Device* is a pointer to the structure of type `t_device`, which contains all the properties of the new device driver. This data must be created and initialized by the user. It must not be modified after being registered at startup of IF/Prolog.

The functions of the device driver are called by IF/Prolog mostly as result of some predicate calls (e.g. `open/3/4`).

Arguments

Device	Pointer to structure of the type `t_device`

Hints

The data structures and functions of device drivers are described in the *IF/Prolog User's Guide* [2].

The function `DeviceCreate()` should be called from the C function `Cboot()`, which is executed at startup of IF/Prolog.

See also

Cboot(), open/3/4, device_control/2, stream_control/2

Release memory space used by IF/Prolog

```
#include "cpro.h"
```

```
BOOLEAN EndProlog(void)
```

The function `EndProlog()` releases the memory space used by IF/Prolog.

The function `EndProlog()` always returns `TRUE`.

Hints

If, after calling `EndProlog()`, a further goal needs to be executed, IF/Prolog must be initialized again with `InitProlog()` and the goal must be parsed with `ParseProlog()`.

Calling `EndProlog()` is not mandatory, but is advisable for reasons of memory space economy.

See also

InitProlog()

Clear error context

```
#include "cpred.h"
```

```
void ErrorContextClear(void)
```

The function `ErrorContextClear()` clears the error context.

Hints

The error context is always cleared before a C predicate is called. If a C predicate returns `FALSE` and the error context is set, IF/Prolog automatically initiates an exception which refers to the C predicate and takes the information in the error context into consideration.

Example

The following simple C function realizes a predicate `is_list/1`, which checks if its argument is a regular list.

```
#include "cpred.h"

static BOOLEAN is_list_1(void)
/*
** is_list( +Term )
*/
{
    TERM tail;

    tail = PrologArg(1);
    while ( TermIsList(tail, NULL, &tail) ) ;

    if ( ErrorContextIsSet() )
    {
         ErrorContextClear(); return FALSE;
    }
    return TRUE;
}

void Cboot(void)
{
    CPRIM("is_list",1,is_list_1);
}
```

See also

ErrorContextGet(), ErrorContextIsSet(), ErrorContextSet()

Query error context

```
#include "cpred.h"
```

```
BOOLEAN ErrorContextGet(ERRORCLASS *Errorclass, STRING
*Parm1, STRING *Parm1, ARITY *Argno, TERM *Culprit)
```

The function `ErrorContextGet()` queries the error context.

If the error context is set, the appropriate error context parameters (see `ErrorContextSet()`) are returned, and the function returns `TRUE`.

If the error context is not set, the function returns `FALSE`. In this case, **Errorclass*, **Parm1*, **Parm2*, **Argno* and **Culprit* are not changed.

For *Errorclass*, *Parm1*, *Parm2*, *Argno* and *Culprit* the constant `NULL` can be passed instead of a pointer to the corresponding data structure. In this case, the corresponding argument is not set.

Arguments

Errorclass	Pointer to an element from the set `ERRORCLASS` (see *cpred.h*)
Parm1, *Parm2*	Pointer to character string or `NULL`
Argno	Pointer to integer or `NULL`
Culprit	Pointer to term variable or `NULL`

Hints

The error context is always cleared before a C predicate is called. If a C predicate returns `FALSE` and the error context is set, IF/Prolog automatically initiates an exception which refers to the C predicate and takes the information in the error context into consideration.

See also

ErrorContextClear(), ErrorContextIsSet(), ErrorContextSet()

Status of the error context

```
#include "cpred.h"
```

```
BOOLEAN ErrorContextIsSet(void)
```

The function `ErrorContextIsSet()` returns `TRUE`, if the error context is set, otherwise it returns `FALSE`.

Hints

The error context is always cleared before a C predicate is called. If a C predicate returns `FALSE` and the error context is set, IF/Prolog automatically initiates an exception which refers to the C predicate and takes the information in the error context into consideration.

Example

The following simple C function realizes a predicate `list_length/2`, which determines the length of a regular list.

```
#include "cpred.h"

static BOOLEAN list_len_2(void)
/*
** list_len( +List, -Length )
*/
{
    TERM tail;
    long length = 0;

    tail = PrologArg(1);
    while ( TermIsList(tail, NULL, &tail) )
        ++length;
    return !ErrorContextIsSet()
    &&     TermUnifyInteger(PrologArg(2), length);
}

void Cboot(void)
{
    CPRIM("list_len",2,list_len_2);
}
```

See also

ErrorContextClear(), ErrorContextGet(), ErrorContextSet()

Set error context

```
#include "cpred.h"
```

```
void ErrorContextSet(ERRORCLASS Errorclass, STRING Parm1,
STRING Parm2, ARITY Argno, TERM Culprit)
```

The function `ErrorContextSet()` is used to set the error context.

Errorclass determines the error class, *Parm1* and *Parm2* the error message, *Argno* the argument number of the term that caused the error, and *Culprit* the term that caused the error.

If the argument number of the erroneous term is not known, *Argno* may be the value `0`. If the erroneous term is not known or does not match the *Argno* term, *Culprit* can be `NULL`.

Depending on *Errorclass*, the following exceptions are created by `ErrorContextSet()`:

`INSTANTIATION_ERROR`
: exception(instantiation_error, [valid_type= *Parm1*, argno= *Argno*, culprit= *Culprit*, goal= *Goal*])

`TYPE_ERROR`
: exception(type_error(*Parm1*, *Culprit*), [argno= *Argno*, goal= *Goal*])

`DOMAIN_ERROR`
: exception(domain_error(*Parm1*, *Culprit*), [argno= *Argno*, goal= *Goal*])

`EXISTENCE_ERROR`
: exception(existence_error(*Parm1*, *Culprit*), [argno= *Argno*, goal= *Goal*])

`PERMISSION_ERROR`
: exception(permission_error(*Parm1*, *Parm2*, *Culprit*), [argno= *Argno*, goal= *Goal*])

`REPRESENTATION_ERROR`
: exception(representation_error(*Parm1*, *Culprit*), [argno= *Argno*, goal= *Goal*])

`EVALUATION_ERROR`
: exception(evaluation_error(*Parm1*), [argno= *Argno*, culprit= *Culprit*, goal= *Goal*])

`RESOURCE_ERROR`
: exception(resource_error(*Parm1*), [argno= *Argno*, culprit= *Culprit*, goal= *Goal*])

`SYNTAX_ERROR`
: exception(syntax_error, [syntax_error= *Parm1*, argno= *Argno*, culprit= *Culprit*, goal= *Goal*])

`SYSTEM_ERROR` or invalid argument
: exception(system_error, [error= *Parm1*, argno= *Argno*, culprit= *Culprit*, goal= *Goal*])

If *Argno* is the value 0, the entry `argno=`*Argno* is not added to the info list for the exceptions.

If *Culprit* is the value `NULL`, the entry `culprit=`*Culprit* is not added to the info list for the exceptions. If *Culprit* is part of the error (first argument of `exception/2`) and is the value `NULL`, the atom `unknown` is used instead of *Culprit*.

The element `goal=`*Goal* is created by IF/Prolog.

Arguments

Errorclass	Element of `ERRORCLASS` (see *cpred.h*)
Parm1, *Parm2*	String
Argno	Integer, $0 \leq$ *Argno* $\leq$ Arity of the C predicate
Culprit	Erroneous term or `NULL`

Hints

The error context is always cleared before a C predicate is called. If a C predicate returns `FALSE` and the error context is set, IF/Prolog automatically initiates an exception which refers to the C predicate and takes the information in the error context into consideration.

Example

The following simple C function realizes a predicate `month/2`, which determines the name of a month according to its number.

```
#include "cpred.h"

static BOOLEAN month_2(void)
/*
** month( +Number, -Name )
*/
{
    long number;
    static const char *names[] = {
        "January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December" };

    if ( TermIsInteger(PrologArg(1), &number) )
    {
        if ( 1 <= number && number <= 12 )
            return TermUnifyAtom(PrologArg(2), names[number-1]);
```

```
            ErrorContextSet(DOMAIN_ERROR, "month", NULL, 1, NULL);
        }
        return FALSE;
    }

    void Cboot(void)
    {
        CPRIM("month",2,month_2);
    }
```

See also

ErrorContextClear(), ErrorContextGet(), ErrorContextIsSet()

Formatted output of Prolog terms and C data objects

```
#include "cpro.h"
```

```
size_t fprintt(FILE *FilePointer, STRING Format, ...)
```

The function `fprintt()` is an extension of the C function `fprintf()` (see [10]) for Prolog terms.

An argument in ... may be either a Prolog term or a normal C data object. The arguments are printed in accordance with the formatting specifications in *Format*.

The output is printed on the stream *FilePointer*.

Arguments

FilePointer	File pointer
Format	Character string, format specification (see `write_formatted/2/3`)
...	Prolog terms or C data objects in accordance with *Format*

See also

outputlen(), printt(), sprintt(), write_formatted/2/3

Initialize IF/Prolog and define the standard media

```
#include "cpro.h"
```

```
BOOLEAN InitProlog(int Number, STRING *ArgumentVector, FILE
*Input, FILE *Output, FILE *Error)
```

The function `InitProlog()` initializes IF/Prolog in the following sequence:

- Memory space is allocated for the Prolog database and for the Prolog stack. Size definitions given in the argument vector *ArgumentVector* are used.
- The internal data structures are initialized.
- The standard media for IF/Prolog are defined. The stream given in the argument *Input* will be used as the standard input (`user_input`), the stream given in the argument *Output* will be used as the standard output (`user_output`) and the stream given in the argument *Error* will be used as the standard error output (`user_error`) of IF/Prolog.
- The C interface and the device drivers are initialized, the function `Cboot()` is called.
- Files are loaded or consulted if required in the argument vector *ArgumentVector*.

Number parameters in the argument vector *ArgumentVector* are processed analogous to the start of IF/Prolog (see `system_parameters/1`).

If the initialization of the IF/Prolog was successful, `InitProlog()` returns `TRUE`, otherwise it returns `FALSE`. If the initialization is not successful and the reason can be determined by IF/Prolog, the global variable `PrologInitError` is set with a string containing an error message.

Arguments

Number	Integer, size of the *ArgumentVector*
ArgumentVector	Vector of pointers to character strings
Input	File pointer, standard input
Output	File pointer, standard output
Error	File pointer, standard error

Hints

The call of this function is necessary for the use of the passive C interface.

See also

EndProlog(), ParseProlog()

Add a complex C function to a module

```
#include "cpred.h"
```

```
void MCPRED(STRING Module, STRING Functor, ARITY Arity,
CPREDFUN Function, size_t Size)
```

The function `MCPRED()` is used to incorporate the predicate *Functor*/*Arity* into IF/Prolog. The predicate is implemented by the complex C function *Function*. A complex function is re-executable, i.e. allows backtracking.

Size specifies the size, in bytes, of the data structure needed for built-in backtracking.

The predicate is assigned to the module *Module*.

Arguments

Module	Character string
Functor	Character string
Arity	Integer, $0 \leq Arity \leq 127$
Function	C Function
Size	Integer

Hints

The function `MCPRED()` should be called from the C function `Cboot()`, which is executed at startup of IF/Prolog.

A module cannot be defined by simply adding C functions. The predicates can only be used as soon as the module interface is created.

See also

Cboot(), CPRED(), CPRIM(), MCPRIM()

Add a simple C function to a module

```
#include "cpred.h"
```

```
void MCPRIM(STRING Module, STRING Functor, ARITY Arity,
CPRIMFUN Function)
```

The function `MCPRIM()` is used to incorporate the predicate *Functor*/*Arity* into IF/Prolog. The predicate is implemented by the simple C function *Function*. A simple function is deterministic, i.e. does not allow backtracking.

The predicate is assigned to the module *Module*.

Arguments

Module	Character string
Functor	Character string
Arity	Integer, $0 \leq Arity \leq 127$
Function	C Function

Hints

The function `MCPRIM()` should be called from the C function `Cboot()`, which is executed at startup of IF/Prolog.

A module cannot be defined by simply adding C functions. The predicates can only be used as soon as the module interface is created.

See also

Cboot(), CPRED(), CPRIM(), MCPRED()

Determine length of formatted output

```
#include "cpro.h"
```

```
size_t outputlen(STRING Format, ...)
```

The function `outputlen()` returns the number of characters that would be printed at the formatted output of the arguments ... with the format *Format*.

An argument ... may be either a Prolog term or a normal C data object. The arguments are printed in accordance with the formatting

Arguments

Format	Character string, format specification (see `write_formatted/2/3`)
...	Prolog terms or C data objects in accordance with *Format*

See also

fprintt(), printt(), sprintt(), write_formatted/2/3

Assignment of syntax error number to syntax error message

```
#include "cpro.h"
```

```
STRING ParseError(int Number)
```

The function `ParseError()` determines the syntax error message associated with *Number*.

The function returns a pointer to a character string containing the error message.

If a *Number* for which no error message is known is specified, the string `unknown_syntax_error` is returned.

The following syntax errors may occur:

Number	Message
1	`.  expected`
2	`) expected`
3	`end of comment */ expected`
4	`illegal character`
5	`start of term expected`
6	`unknown stand alone character`
7	`operator expected`
8	`in/postfix operator or end of term expected`
9	`too large constant`
10	`| ] or , expected`
11	`} expected`
12	`' or " or ` expected`
13	`, or ) expected`
14	`illegal number`
15	`precedence error`
16	`] expected`
17	`interrupt`
18	`i/o error`
19	`illegal escape sequence`
20	`illegal character code`
21	`exceeded max_arity`

Arguments

Number	Integer

Hints

The structure component `t_parse_context.pc_errno` is set by the function `ParseProlog()` with the error number when a syntax error is determined.

See also

ParseProlog(), InitProlog(), EndProlog(), syntax_error/2

Parse a goal string and store the information

```
#include "cpro.h"
```

```
BOOLEAN ParseProlog(STRING GoalAtom, t_parse_context
*Context)
```

The function `ParseProlog()` prepares the execution of a Prolog goal.

The character string *String* is parsed in accordance with the Prolog syntax (see the section 9 on *Prolog syntax*). Parsing ends at the first period (`.`) which cannot be part of the term syntax. The characters after the `.` are not used.

If the parsing of the string was successful, `ParseProlog()` returns `TRUE`, otherwise it returns `FALSE`.

The information about the goal is stored in the structure pointed at by *Context*. The following structure components of *Context* are set when parsing is successful.

Component	Meaning
Context`->pc_term`	Prolog term
Context`->pc_varlist`	List of Prolog variables
Context`->pc_position`	Position after the end
Context`->pc_errno`	0

The following structure components of *Context* are set when a syntax error is determined:

Component	Meaning
Context`->pc_position`	Position of the syntax error
Context`->pc_errno`	Syntax error number

Arguments

GoalAtom	Character string
Context	Pointer to structure where information about the Prolog goal is stored

Hints

The parsed Prolog goal can be processed by `PrologOpen()` or `QueryCreate()`.

See also

ParseProlog(), QueryCreate(), ParseError(), parse_atom/6

Formatted output of Prolog terms and C data objects

```
#include "cpro.h"
```

```
size_t printt(STRING Format, ...)
```

The function `printt()` is an extension of the C function `printf()` (see [10]) for Prolog terms.

An argument in ... may be either a Prolog term or a normal C data object. The arguments are printed in accordance with the formatting specifications in *Format*.

The output is printed on the standard output.

Arguments

Format	Character string, format specification (see `write_formatted/2/3`)
...	Prolog terms or C data objects in accordance with *Format*

See also

outputlen(), fprintt(), sprintt(), write_formatted/2/3

Access predicate arguments

```
#include "cpred.h"
```

```
TERM PrologArg(ARITY Argno)
```

The function `PrologArg()` returns the *Argno*th argument of the active C predicate.

Arguments

Argno	Integer, $1 \leq Argno \leq$ arity of the C predicate.

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Argno* **: argno out of range**
The argument *Argno* is not within the prescribed range of values.

Example

The following simple C function realizes a predicate `counter/1`, which returns an integer number incremented by 1 at each call (starting at 0).

```
#include "cpred.h"

BOOLEAN counter_1(void)
/*
** counter( -I )
*/
{
    static long counter = 0;

    return TermUnifyInteger(PrologArg(1), counter++);
}

void Cboot(void)
{
    CPRIM("counter",1,counter_1);
}
```

See also

TermArg()

Deactivate the current Prolog goal

```
#include "cpro.h"
```

```
void PrologClose(t_cursor ActGoal)
```

The function `PrologClose()` releases the current Prolog goal (see functions `PrologOpen()` or `QueryOpen()`) and all appropriate resources.

The goal which was current before the creation of *ActGoal* becomes the current goal again.

Arguments

ActGoal	Structure, information on the current Prolog goal

Hints

The structure *ActGoal* is undefined after the call of `PrologClose()`.

See also

PrologOpen(), QueryOpen(), PrologFetch()

Enable epilog

```
#include "cpred.h"
```

```
void PrologEnableEpilog(void)
```

The function `PrologEnableEpilog()` defines that a complex C function has an epilog. This epilog will be called as soon as the complex C function can find no more solutions.

Manipulations of terms is not allowed during execution of an epilog. The term arguments of the complex C function are not defined during the execution of the epilog.

The function `PrologEnableEpilog()` must be called at the first call and not during backtracking of the complex C function. This call must take place before the first unification.

Hints

A detailed description of macros used in the example to get a more clear implementation of complex C-predicates (`COROUTINE` etc.) can be found in the *IF/Prolog User's Guide* [2].

Example

The predicate `readchar(`D`,`Z`)` opens at the first call the file D, reads the first character and unifies it with Z. With backtracking each next character will be read in and unified with Z. The file will be closed at the end of the file or when backtracking is cut off (see `!/0`).

```
#include <stdio.h>
#include "cpred.h"

typedef struct { FILE *fp; } readstate;

COROUTINE(readchar, readstate, state)
        RESUME(1,label1);
        HASEPILOG;
BEGIN
        STRING filename;
        int c;

        if ( !TermIsAtom(PrologArg(1),&filename) ) FAIL;
        if ( (state->fp = fopen(filename,"r")) == NULL ) FAIL;

        PrologEnableEpilog();
```

```
        while ( (c = fgetc(state->fp)) != EOF )
        {
                if ( TermUnifyInteger(PrologArg(2),(long)c) )
                        DETACH(1,label1);
                else    PrologUndo();
        }

EPILOG:
        fclose(state->fp);
END

void Cboot(void)
{
        CPRED("readchar",2,readchar,sizeof(readstate));
}
```

See also

!/0

Query error

```
#include "cpro.h"
```

```
BOOLEAN PrologError(t_cursor ActGoal, TERM *Term)
```

The function `PrologError()` returns `TRUE`, when an error has occurred during the execution of the goal *ActGoal* with `PrologFetch()`.

An error has occurred when during the execution of *ActGoal* a context jump (e.g. implicitly triggered by an exception) is initiated and no matching context can be found.

The term pointed at by *Term* is set to the error term (see `throw/1`) which contains information about the error.

Arguments

ActGoal	Structure, information on the current Prolog goal
Term	Pointer to term variable

Hints

When an error has occurred during the execution of *ActGoal*, the next activation of *ActGoal* with `PrologFetch()` will restart the execution of the goal, i.e. the first solution will then be found.

See also

PrologClose(), PrologFetch(), PrologOpen()

Find solutions for a goal

```
#include "cpro.h"
```

```
BOOLEAN PrologFetch(t_cursor ActGoal)
```

The function `PrologFetch()` searches for a solution for the current Prolog goal *ActGoal*.

If a solution can be found, `PrologFetch()` returns `TRUE`. Otherwise it returns `FALSE`.

When a solution has been found, a subsequent call of `PrologFetch()` will lead to the next solution, e.g. backtracking takes place.

If no solution has been found or if an error did occur (see `PrologError()`), a subsequent call of `PrologFetch()` will restart the query and will therefore lead to the first solution again.

The following steps can be followed after IF/Prolog has been initialized with `InitProlog()`:

1. Call of `ParseProlog()` in order to get a goal term.
2. By means of `PrologOpen()` this goal becomes the current goal and is therefore ready for execution.
3. One or more calls of `PrologFetch()` execute the goal. The solutions may be analyzed with `PrologGoal()` and the functions of the active C interface and may be processed further.
4. The current goal is deactivated with `PrologClose()`.

Arguments

ActGoal	Structure, information on the current Prolog goal

Hints

Solutions can not be simultaneously searched for more than one goal. However, nested execution of queries is possible.

See also

ParseProlog(), PrologClose(), PrologError(), PrologOpen(), QueryOpen()

Prolog goal as term

```
#include "cpro.h"
```

```
TERM PrologGoal(t_cursor ActGoal)
```

The function `PrologGoal()` returns the current goal *ActGoal* as term.

After a successful execution this term contains the *results* as bound arguments.

Arguments

ActGoal	Structure, information on the current Prolog goal

See also

PrologFetch(), PrologOpen(), QueryOpen()

Get floating-point number from variable

```
#include "cpro.h"
```

```
BOOLEAN PrologIsFloat(t_cursor ActGoal, STRING VarName,
double *Value)
```

The function `PrologIsFloat()` checks, if the current Prolog goal *ActGoal* contains a Prolog variable with the name *VarName* and if this variable is bound to a floating-point number.

If such a variable is found, the function stores the value of the number in **Value* and returns `TRUE`.

Otherwise the function returns `FALSE`.

Arguments

ActGoal	Structure, information on the current Prolog goal
VarName	Character string, name of a Prolog variable
Value	Pointer to a floating-point number

See also

PrologIsInteger(), PrologIsString(), PrologIsTerm()

Get integer from variable

```
#include "cpro.h"
```

```
BOOLEAN PrologIsInteger(t_cursor ActGoal, STRING VarName,
long *Value)
```

The function `PrologIsInteger()` checks, if the current Prolog goal *ActGoal* contains a Prolog variable with the name *VarName* and if this variable is bound to an integer.

If such a variable is found, the function stores the value of the number in **Value* and returns `TRUE`.

Otherwise the function returns `FALSE`.

Arguments

ActGoal	Structure, information on the current Prolog goal
VarName	Character string, name of a Prolog variable
Value	Pointer to an integer

See also

PrologIsFloat(), PrologIsString(), PrologIsTerm()

Get character string from variable

```
#include "cpro.h"
```

```
BOOLEAN PrologIsString(t_cursor ActGoal, STRING VarName,
STRING *Name)
```

The function `PrologIsString()` checks, if the current Prolog goal *ActGoal* contains a Prolog variable with the name *VarName* and if this variable is bound to a Prolog atom.

If such a variable is found, the function stores a pointer to the name of the atom in **Name* and returns `TRUE`.

Otherwise the function returns `FALSE`.

Arguments

ActGoal	Structure, information on the current Prolog goal
VarName	Character string, name of a Prolog variable
Name	Pointer to pointer to character

See also

PrologIsInteger(), PrologIsFloat(), PrologIsTerm()

Get Prolog term from variable

```
#include "cpro.h"
```

```
BOOLEAN PrologIsTerm(t_cursor ActGoal, STRING VarName, TERM *Term)
```

The function `PrologIsTerm()` checks, if the current Prolog goal *ActGoal* contains a Prolog variable with the name *VarName*.

If such a variable is found, the function stores the term bound to this variable in **Term* and returns `TRUE`.

Otherwise the function returns `FALSE`.

Arguments

ActGoal	Structure, information on the current Prolog goal
VarName	Character string, name of a Prolog variable
Term	Pointer to term variable

See also

PrologIsInteger(), PrologIsFloat(), PrologIsString()

Get Prolog module

```
#include "cpro.h"
```

```
MODULE PrologModule(STRING Name)
```

The function `PrologModule()` checks if a module with the name *Name* exists and returns it.

The function returns `NULL` if the module *Name* is not defined within IF/Prolog.

Arguments

Name	Character string

Hints

The predefined modules `system` and `user` are accessible by the global variables `System-Module` and `UserModule`.

Module specifications are used to indicate explicitly the module context in which a predicate is to be executed.

See also

PrologFetch(), PrologOpen(), QueryCreate()

Create current Prolog goal

```
#include "cpro.h"
```

```
t_cursor PrologOpen(MODULE Module, TERM Goal, TERM VarList)
```

The function `PrologOpen()` creates an active Prolog goal. This Prolog goal also becomes the current Prolog goal.

The goal to be prepared for execution is defined by the goal term *Goal* and *VarList*, the list of the variables in *Goal*. This can be the result of a preceding call of `ParseProlog()`.

The goal can then be executed with `PrologFetch()` in the module context *Module*.

The function returns the structure `t_cursor` with the information about the state of the active Prolog goal. This information will be used as reference to the current goal.

A goal that was the current goal before the call of `PrologOpen()` remains an actual goal, but will only be accessible after the execution of the new current goal has been terminated (see `PrologClose()`). Therefore, nested execution of queries is possible.

Arguments

Module	Structure (see `PrologModule()`)
Goal	Term
VarList	Term, list of variables in *Goal*

See also

PrologModule(), PrologClose(), PrologError(), PrologFetch(), QueryOpen(), ParseProlog()

Undo unification

```
#include "cpred.h"
```

```
void PrologUndo(void)
```

The function `PrologUndo()` cancels (undoes) all term manipulations (see `TermUnify...()`, `TermMake...()`, ...) in the current call of a C function.

All terms constructed since the start of a C function are now void and all term unifications are undone.

The function `PrologUndo()` is necessary, when after a failing unification, a C function is not immediately terminated with `return FALSE` or `FAIL`.

Hints

A detailed description of macros used in the example to get a more clear implementation of complex C-predicates (`COROUTINE` etc.) can be found in the *IF/Prolog User's Guide* [2].

Example

The second argument L of the predicate `element_of(`E`,`L`)` shall be a list of Prolog terms. By backtracking, the first argument E is unified with each term of the list L.

```
#include "cpred.h"

typedef struct { size_t no; } index;

static BOOLEAN
proper_list(TERM list)
{
        TERMCONTEXT context = TermContext();
        while ( TermIsList(list,NULL,&list) ) ;
        TermCollect(context,NULL);
        return !ErrorContextIsSet();
}

static BOOLEAN
nth_element(size_t n, TERM list, TERM *tm)
{
        TERMCONTEXT context = TermContext();
        TERM element;
```

```
        size_t i;

        for ( i = 0; i < n; ++i )
        {
                if ( !TermIsList(list,&element,&list) )
                {
                        TermCollect(context,NULL);
                        return FALSE;
                }
        }

        TermCollect(context,&element,NULL);
        *tm = element;
        return TRUE;
}

COROUTINE(element_of, index, state)
        RESUME(1,label1);
        HASEPILOG;
BEGIN

        state->no = 0;
        if ( !proper_list(PrologArg(2)) ) FAIL;

        while ( nth_element(++state->no,PrologArg(2),&tm) )
        {
                if ( TermUnify(PrologArg(1),tm) )
                        DETACH(1,label1);
                else    PrologUndo();
        }

END

void Cboot(void)
{
        CPRED("element_of",2,element_of,sizeof(index));
}
```

```
#include "cpro.h"
```

```
t_query QueryCreate(MODULE Module, TERM Goal, TERM VarList)
```

The function `QueryCreate()` creates a passive Prolog goal. The function returns the structure `t_query` with the information about the active Prolog goal. This data structure will be needed for future reference to the passive Prolog goal.

A passive Prolog goal is not executable after creation. It can be later repeatedly activated by `QueryOpen()` and then executed.

The passive goal is defined by the goal term *Goal* and *VarList*, the list of the variables in *Goal*. This can be the result of a preceding call of `ParseProlog()`.

The goal will be executed in the module context *Module*.

Arguments

Module	Structure (see `PrologModule()`)
Goal	Term
VarList	Term, list of variables in *Goal*

See also

ParseProlog(), PrologModule(), QueryDispose(), QueryOpen()

Release passive Prolog goal

```
#include "cpro.h"
```

```
void QueryDispose(t_query Query)
```

The function `QueryDispose()` releases a passive Prolog goal and all resources used by the goal.

Arguments

Query	Structure, information on the passive Prolog goal

Hints

After the call of `QueryDispose()` the data structure *Query* is undefined.

See also

QueryCreate(), QueryOpen()

Activate a passive Prolog goal

```
#include "cpro.h"
```

```
t_cursor QueryOpen(t_query Query)
```

The function `QueryOpen()` activates a passive Prolog goal. This Prolog goal also becomes the current Prolog goal.

The goal can then be executed with `PrologFetch()`.

The function returns the structure `t_cursor` containing the information about the state of the active Prolog goal. This information will be used as reference to the current goal.

A goal that was the current goal before the call of `QueryOpen()` remains an actual goal, but will only be accessible after the execution of the new current goal has been terminated (see `PrologClose()`). Therefore nested execution of queries is possible.

Arguments

Query	Structure, information on the passive Prolog goal

See also

PrologClose(), QueryCreate(), QueryDispose(), PrologFetch()

Formatted output of Prolog terms and C data objects

```
#include "cpro.h"
```

```
size_t sprintt(char *Buffer, STRING Format, ...)
```

The function `sprintt()` is an extension of the C function `sprintf()` (see [10]) for Prolog terms.

An argument in ... may be either a Prolog term or a normal C data object. The arguments are printed in accordance with the formatting specifications in *Format*.

The output is printed on the character string pointed at by *Buffer*.

Arguments

Buffer	Pointer to character string
Format	Character string, format specification (see `write_formatted/2/3`)
...	Prolog terms or C data objects in accordance with *Format*

See also

outputlen(), fprintt(), printt(), write_formatted/2/3

Flush Prolog output buffer

```
#include "device.h"
```

```
BOOLEAN StreamFlush(t_stream Stream, long *Number)
```

The function `StreamFlush()` can be used by the control functions of a device driver in order to cause IF/Prolog to flush its output buffer.

The argument *Stream* identifies the input/output stream and has been given to the control function as argument.

The integer pointed at by *Number* is set to the number of characters which were in the buffer prior to flushing. If the constant `NULL` has been passed for the argument *Number*, the argument is not set.

Arguments

Stream	Stream
Number	Pointer to long

Hints

The data structures and functions of device drivers are described in the *IF/Prolog User's Guide* [2].

See also

DeviceCreate(), flush_output/0/1

Query stream mode

```
#include "device.h"
```

```
t_iomode StreamMode(t_stream Stream)
```

The function `StreamMode()` returns the mode of a stream.

The argument *Stream* identifies the input/output stream and has been given to the control function as argument.

By means of this function, the device driver does not necessarily have to store all the mode information of every opened stream.

Arguments

Stream	Stream

Hints

The data structures and functions of device drivers are described in the *IF/Prolog User's Guide* [2].

See also

DeviceCreate(), stream_property/2

Add to a Prolog list

```
#include "cpred.h"
```

```
BOOLEAN TermAddList(TERM *Tail, TERM Term)
```

The function `TermAddList()` is used to construct lists. The term *Term* is appended to a list.

This function unifies the term **Tail* with a list, whose head is the term *Term* and tail is a variable.

If unification is successful, **Tail* is supplied with the new list tail and the function returns `TRUE`.

Otherwise the function returns `FALSE`.

Arguments

Tail	Pointer to term variable
Term	Term variable

Example

The following simple C function realizes a predicate `make_list/2`, which builds a list of five subsequent integers starting with the number given by the first argument.

```
#include "cpred.h"

BOOLEAN make_list_2(void)
/*
** make_list( +Start, -List )
*/
{
    TERM list, tail;
    long start, i;

    if (!TermIsInteger(PrologArg(1),&start))
        return FALSE;
    list = TermOpenList(&tail);
    for (i=0;i<5;i++)
        TermAddList(&tail,TermMakeInteger(start+i));
    TermCloseList(tail);

    return TermUnify(PrologArg(2),list);
```

```
}

void Cboot(void)
{
    CPRIM("make_list",2,make_list_2);
}
```

See also

TermOpenList(), TermCloseList()

Access structure arguments

```
#include "cpred.h"
```

```
TERM TermArg(ARITY Argno, TERM Term)
```

The function `TermArg()` returns the *Argno*th argument of the structure *Term*.

If *Term* is not a structure or *Argno* is not within the valid range of values, IF/Prolog is exited with a system error message.

Arguments

Argno	Integer, $1 \leq$ *Argno* $\leq$ Arity of *Term*.
Term	Term variable

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Argno* **: argno out of range**
The argument *Argno* is not within the prescribed range of values.

implementation_error: TermArg : compound term expected
The argument TermArg is not a structure.

See also

PrologArg(), TermDecompose()

Close a Prolog list

```
#include "cpred.h"
```

```
BOOLEAN TermCloseList(TERM Tail)
```

The function `TermCloseList()` closes a list opened by `TermOpenList()` and built by `TermAddList()`.

Technically this function unifies *Tail* with an empty list.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Tail	Term variable

Example

See example at `TermAddList()`.

See also

TermAddList(), TermOpenList(), TermUnifyNil()

Release term variables

```
#include "cpred.h"
```

```
void TermCollect(TERMCONTEXT Context, ...)
```

```
void TermCollectSequence(TERMCONTEXT Context, size_t N,
TERM *TermList)
```

These functions release all term variables that were generated since *Context* was determined, with the exception of the pointers to the term variables listed in ... or *TermList*.

The function `TermCollect()` contains a list of optional parameters (...). The list must end with `NULL`. The optional parameters are pointers to term variables that are not to be released.

A vector *TermList* with *N* terms as the parameters is passed to the function `TermCollectSequence()`. The terms included in *TermList* are not to be released.

Arguments

Context	Term context
...	Prolog terms, the last argument must be `NULL`
N	Integer, number of terms in vector *TermList*
TermList	Vector consisting of the terms

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: TermCollect : illegal context
The argument TermCollect is not a term context generated with `TermContext()`.

implementation_error: TermCollectSequence : illegal context
The argument TermCollectSequence is not a term context generated with `TermContext()`.

Example

The following C function displays a term in LISP notation. The internally user term variables are released as soon as possible in order to save memory space.

```
#include "cpred.h"

static void display_lisp(TERM t)
```

```
{
    TERMINFO info;
    TERMCONTEXT context;
    ARITY argno;

    switch ( TermDecompose(t, &info) )
    {
    case TERM_COMPOUND:
        printt("(%s", info.compound.functor);
        context = TermContext();
        for ( argno = 1; argno <= info.compound.arity; ++argno )
        {
            printt(" ");
            display_term(TermArg(argno, t));
        }
        TermCollect(context, NULL);
        printt(")");
        break;
    default:
        printt("%q", t);
    }
}
```

The following simple C function realizes a predicate `write_lisp_list/1`, which prints elements of a list in LISP notation using the C function `display_lisp()`.

```
BOOLEAN write_lisp_list_1(void)
/*
** write_lisp_list( +Term )
*/
{
    TERM tail;

    tail = PrologArg(1);
    while (TermIsList(tail,&head,&tail))
        display_lisp(head);
    return ErrorContextIsSet();
}

void Cboot(void)
{
    CPRIM("write_lisp_list",1,write_lisp_list_1);
}
```

See also

TermContext()

Determine current term context

```
#include "cpred.h"
```

```
TERMCONTEXT TermContext(void)
```

The function `TermContext()` determines the current term context.

The term context comprises all term variables managed by the C interface when `TermContext()` is called.

Example

See example at `TermCollect()`.

See also

TermCollect(), TermCollectSequence()

Classify term

```
#include "cpred.h"
```

```
TERMTYPE TermDecompose(TERM Term, TERMINFO *Info)
```

The function `TermDecompose()` returns the type of the term *Term*. In addition, the *Info* argument is supplied with information that is set depending of the term type.

The function returns the following values:

TERM_VAR
: The term is an uninstantiated Prolog variable.
Info`->varno` is supplied with a variable number. Two variables possess the same number if they are identical.

  Note: *Info*`->varno` loses its validity when a function is called which generates new terms or evaluates arithmetic expressions.

TERM_ATOM
: The term is a Prolog atom.
Info`->name` is supplied with the name of the atom.

  Note: *Info*`->name` loses its validity when no other term containing an atom or functor with this name can be accessed.

TERM_INTEGER
: The term is an integer within in the range of values for a C variable of the type `signed long`.
Info`->ivalue` is supplied with the value of the number.

TERM_FLOAT
: The term is a floating-point number.
Info`->rvalue` is supplied with the value of the number.

TERM_COMPOUND
: The term is a structure.
Info`->compound.functor` is supplied with the name of the structure.
Info`->compound.arity` is supplied with the arity of the structure.

  Note: *Info*`->compound.functor` loses its validity when no other term containing an atom or functor with this name can be accessed.

TERM_OTHER
: The term has a type not listed above.
The contents of the structure *Info* is undefined.

Arguments

Term	Term variable
Info	Pointer to a structure which is to receive type-dependent information on a term (see `cpred.h`).

Hints

Future versions of IF/Prolog will extend the definitions of the data types TERMTYPE and TERMINFO. If the `TermDecompose()` function is used, the user should make sure that the function can also return a value that is not currently defined as the result.

Example

The following simple C function realizes a predicate `type_of/3`, which gives the type and more information about a term.

```
#include "cpred.h"

BOOLEAN type_of_3(void)
/*
** type_of( +Term, -Type, -Info )
*/
{
    TERMINFO terminfo;
    TERM info;
    STRING typename;

    info = PrologArg(1);
    switch ( TermDecompose(PrologArg(1), &terminfo) )
    {
    case TERM_VAR:
        typename = "var";
        break;
    case TERM_INTEGER:
        typename = "integer";
        break;
    case TERM_FLOAT:
        typename = "float";
        break;
    case TERM_ATOM:
        typename = "atom";
        break;
    case TERM_COMPOUND:
        typename = "compound";
```

```
            info = TermMakeCompound("/", 2,
                           TermMakeAtom(terminfo.compound.functor),
                           TermMakeInteger(terminfo.compound.arity));
            break;
        default:
            typename = "unknown";
        }

        return TermUnifyAtom(PrologArg(2), typename)
        &&     TermUnify(PrologArg(3), info);
    }

    void Cboot(void)
    {
        CPRIM("type_of",3,type_of_3);
    }
```

See also

TermType(), TermArg()

Test for atom

```
#include "cpred.h"
```

```
BOOLEAN TermIsAtom(TERM Term, STRING *Name)
```

The function `TermIsAtom()` tests whether the term *Term* is an atom.

If it is, the function sets **Name* to point to the name of the atom and returns `TRUE`.

Otherwise it sets the error context and returns `FALSE`. In this case, **Name* is not changed.

Arguments

Term	Term variable
Name	Pointer to a pointer to character string

Exceptions

instantiation_error
: The argument *Term* must not be a variable, but a variable was specified.

type_error(atom)
: The argument *Term* must be an atom, but is a term of another type.

Hints

The pointer **Name* loses its validity when no term containing an atom or functor with this name can be accessed.

See also

TermMakeAtom()

Test for structure

```
#include "cpred.h"
```

```
BOOLEAN TermIsCompound(TERM Term, STRING Functor, ARITY Arity, ...)
```

The function `TermIsCompound()` tests whether the term *Term* is a structure with the functor *Functor* and the arity *Arity*.

If it is, all the pointers to term variables made available in ... are initialized with term variables which contain the arguments of the structure *Term*. The function then returns `TRUE`.

If *Term* is not a structure, or its functor and arity do not match *Functor* and *Arity* respectively, the function sets the error context and returns `FALSE`.

Atoms are treated like structures with the arity `0`.

Arguments

Term	Term variable
Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$
...	Pointers to term variables

Exceptions

instantiation_error
The argument *Term* must not be a variable, but a variable was specified.

type_error(atom_or_compound)
The argument *Term* must be an atom or a structure, but is a term of another type.

domain_error(compound)
The argument *Term* is a structure, but its functor or arity are not correct.

The following error causes IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

Example

The following simple C function realizes a predicate `indicator/1`, which checks if a term is a predicate indicator, i.e. a term of the form *Functor*/*Arity*.

```
#include "cpred.h"

BOOLEAN indicator_1(void)
/*
** indicator( +Term )
*/
{
    TERM name, arity;
    long number;
    STRING string;

    return TermIsCompound(PrologArg(1),"/",2,&name,&arity)
    &&     TermIsAtom(name,&string)
    &&     TermIsInteger(arity,&number);
}

void Cboot(void)
{
    CPRIM("indicator",1,indicator_1);
}
```

See also

TermIsFunctor(), TermIsUniversal(), TermMakeCompound()

Test for floating-point number

```
#include "cpred.h"
```

```
BOOLEAN TermIsFloat(TERM Term, double *Value)
```

The function `TermIsFloat()` tests whether the term *Term* is a number that can be represented as a floating-point number.

If it is, the function stores the value of the number in **Value* and returns `TRUE`.

Otherwise it sets the error context and returns `FALSE`. In this case, **Value* is not changed.

Arguments

Term	Term variable
Value	Pointer to a floating-point number

Exceptions

instantiation_error
: The argument *Term* must not be a variable, but a variable was specified.

type_error(real)
: The argument *Term* must be a number, whose value can be converted implicitly into a floating-point number, but is a term of another type.

evaluation_error(overflow)
: The argument *Term* is a number for which an overflow occurred during its implicit conversion into a floating-point number.

See also

TermIsFloatExpression(), TermIsInteger(), TermMakeFloat()

Test for floating-point expression

```
#include "cpred.h"
```

```
BOOLEAN TermIsFloatExpression(TERM Term, double *Value)
```

The function `TermIsFloatExpression()` tests whether the term *Term* is an expression, whose result can be represented as a floating-point number.

If it is, the function stores the value of the number in **Value* and returns `TRUE`.

Otherwise it sets the error context and returns `FALSE`. In this case, **Value* is not changed.

Arguments

Term	Term variable
Value	Pointer to a floating-point number

Exceptions

instantiation_error
: The argument *Term* or a subterm must not be a variable, but a variable was specified.

type_error(number)
: The argument *Term* must be a number or an arithmetic expression, whose value can be converted implicitly into a floating-point number, but is a term of another type.

evaluation_error(overflow)
: The argument *Term* is a number for which an overflow occurred during its implicit conversion into a floating-point number.

See also

TermIsFloat(), TermIsIntegerExpression(), TermMakeFloat()

Test for structure

```
#include "cpred.h"
```

```
BOOLEAN TermIsFunctor(TERM Term, STRING Functor, ARITY Arity)
```

The function `TermIsFunctor()` tests whether the term *Term* is a structure with the functor *Functor* and the arity *Arity*.

If it is, the function returns `TRUE`.

If *Term* is not a structure, or its functor and arity do not match *Functor* and *Arity* respectively, the function sets the error context and returns `FALSE`.

Atoms are treated like structures with the arity `0`.

Arguments

Term	Term variable
Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

instantiation_error
The argument *Term* must not be a variable, but a variable was specified.

type_error(atom_or_compound)
The argument *Term* must be an atom or a structure, but is a term of another type.

domain_error(compound)
The argument *Term* is a structure, but its functor or arity are not correct.

The following error causes IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

Example

The following simple C function realizes a predicate `assignment/1`, which checks if its argument is a term of the form *Term1*=*Term2*:

```
#include "cpred.h"

BOOLEAN assignment_1(void)
/*
```

```
** assignment( +Term )
*/
{
    return TermIsFunctor(PrologArg(1),"=",2);
}

void Cboot(void)
{
    CPRIM("assignment",1,assignment_1);
}
```

See also

TermIsCompound(), TermIsUniversal(), TermMakeFunctor()

Test for integer

```
#include "cpred.h"
```

```
BOOLEAN TermIsInteger(TERM Term, long *Value)
```

The function `TermIsInteger` tests whether the term *Term* is an integer within the range of values for a C variable of the type `signed long`.

If it is, the function stores the value of the number in **Value* and returns `TRUE`.

Otherwise it sets the error context and returns `FALSE`. In this case, **Value* is not changed.

Arguments

Term	Term variable
Value	Pointer to an integer number

Exceptions

instantiation_error
The argument *Term* must not be a variable, but a variable was specified.

type_error(integer)
The argument *Term* must be an integer, but is a term of another type.

domain_error(single_precision_integer)
The argument *Term* must be an integer with machine precision, but a big number was specified.

See also

TermIsIntegerExpression(), TermIsFloat(), TermMakeInteger()

Test for integer expression

```
#include "cpred.h"
```

```
BOOLEAN TermIsIntegerExpression(TERM Term, long *Value)
```

The function `TermIsIntegerExpression()` tests whether the term *Term* is an expression, whose result is an integer within the range of values for a C variable of the type `signed long`.

If it is, the function stores the value of the number in **Value* and returns `TRUE`.

Otherwise it sets the error context and returns `FALSE`. In this case, **Value* is not changed.

Arguments

Term	Term variable
Value	Pointer to an integer number

Exceptions

instantiation_error
: The argument *Term* or a subterm must not be a variable, but a variable was specified.

type_error(integer)
: The argument *Term* must be an integer or an integer expression, but is a term of another type, or the result of the evaluation of the expression is not an integer.

See also

TermIsInteger(), TermIsFloatExpression(), TermMakeInteger()

Test for list

```
#include "cpred.h"
```

```
BOOLEAN TermIsList(TERM Term, TERM *Head, TERM *Tail)
```

The function `TermIsList()` tests whether the term *Term* is a non-empty list.

If it is, *Head* is initialized with the list head and *Tail* with the list tail (provided that *Tail* and *Head* are not `NULL`), and the function returns `TRUE`.

If the term is the empty list, the function returns `FALSE`.

If the term is not a list, the function sets the error context and returns `FALSE`.

Arguments

Term	Term variable.
Head, *Tail*	Pointer to term variables

Exceptions

instantiation_error
The argument *Term* must not be a variable, but a variable was specified.

type_error(list)
The argument *Term* must be a list, but is a term of another type.

Example

The following simple C function realizes a predicate `list_sum/2`, which sums up all elements of a list of integers.

```
#include "cpred.h"

BOOLEAN list_sum_2(void)
/*
** list_sum( +List, -Sum )
*/
{
    long value, total;
    TERM head, tail;

    tail = PrologArg(1);
    while (TermIsList(tail,&head,&tail)
```

```
        &&     TermIsInteger(head,&value))
            total += value;

        return !ErrorContextIsSet()
        &&     TermUnifyInteger(PrologArg(2),total);
    }

    void Cboot(void)
    {
        CPRIM("list_sum",2,list_sum_2);
    }
```

See also

TermMakeList()

Test for empty list

```
#include "cpred.h"
```

```
BOOLEAN TermIsNil(TERM Term)
```

The function `TermIsNil()` tests whether the term *Term* is the empty list.

If it is, the function returns `TRUE`.

Otherwise it sets the error context and returns `FALSE`.

Arguments

Term	Term variable

Exceptions

instantiation_error
: The argument *Term* must not be a variable, but a variable was specified.

type_error(list)
: The argument *Term* must be a list, but is a term of another type.

domain_error(list)
: The argument *Term* must be an empty list, but is not empty.

See also

TermIsList(), TermMakeNil()

Test for structure

```
#include "cpred.h"
```

```
BOOLEAN TermIsUniversal(TERM Term, STRING Functor, ARITY
Arity, TERM Args[])
```

The function `TermIsUniversal()` tests whether the term *Term* is a structure with the functor *Functor* and the arity *Arity*.

If it is, the function initializes the term variables in the vector *Args* with the term variables contained in the arguments of the structure *Term*. The function returns `TRUE`.

If *Term* is not a structure, or its functor and arity do not match *Functor* and *Arity* respectively, the function sets the error context and returns `FALSE`.

Atoms are treated like structures with the arity `0`.

Arguments

Term	Term variable
Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$
Args	Vector consisting of term variables

Exceptions

instantiation_error
: The argument *Term* or a subterm must not be a variable, but a variable was specified.

type_error(atom_or_compound)
: The argument *Term* must be an atom or a structure, but is a term of another type.

domain_error(compound)
: The argument *Term* is a structure, but its functor or arity are not correct.

The following error causes IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
: The argument *Arity* is not within the range 0..127.

Example

The following simple C function realizes a predicate `indicator/1`, which checks if its argument is a predicate indicator and returns name and arity.

```
#include "cpred.h"

BOOLEAN indicator_3(void)
/*
** indicator( +Term, -Name, -Arity )
*/
{
    TERM args[2];
    long number;
    STRING string;

    return TermIsUniversal(PrologArg(1),"/",2,args)
    &&     TermIsAtom(args[0],&string)
    &&     TermIsInteger(args[1],&number)
    &&     TermUnify(PrologArg(2),args[0])
    &&     TermUnify(PrologArg(3),args[1]);
}

void Cboot(void)
{
    CPRIM("indicator",3,indicator_3);
}
```

See also

TermIsCompound(), TermIsFunctor(), TermMakeUniversal()

Test for variable

```
#include "cpred.h"
```

```
BOOLEAN TermIsVar(TERM Term)
```

The function `TermIsVar()` tests whether the term *Term* is a Prolog variable.

If it is, the function returns `TRUE`.

Otherwise it sets the error context and returns `FALSE`.

Arguments

Term	Term variable

Exceptions

type_error(variable)
: The argument *Term* must be a variable, but is a term of another type.

See also

TermMakeVar()

Generate a Prolog atom

```
#include "cpred.h"
```

```
TERM TermMakeAtom(STRING Name)
```

The function `TermMakeAtom()` returns a Prolog atom with the name *Name*.

Arguments

Name	Name of the atom

Exceptions

The following error causes IF/Prolog to be exited:

implementation_error: TermMakeAtom : out of memory
: The function `TermMakeAtom()` could not be executed successfully due to a lack of memory.

See also

TermIsAtom()

Generate a Prolog structure

```
#include "cpred.h"
```

```
TERM TermMakeCompound(STRING Functor, ARITY Arity, ...)
```

The function `TermMakeCompound()` generates a Prolog structure with the name *Functor*, the arity *Arity* and the arguments ... and returns it as the result.

If one of the term variables in ... is `NULL`, a Prolog variable is generated instead and is used as the structure argument.

Arguments

Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$
...	*Arity* term variables

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

implementation_error: TermMakeCompound : out of memory
The function `TermMakeCompound()` could not be executed successfully due to a lack of memory.

Example

The following simple C function realizes a predicate `make_indicator/3`, which makes a predicate indicator from given name and arity.

```
#include "cpred.h"

BOOLEAN make_indicator_3(void)
/*
** make_indicator( +Functor, +Arity, -Indicator )
*/
{
    STRING string;
    long value;
    TERM functor = PrologArg(1);
```

```
    TERM arity = PrologArg(2);

    return TermIsAtom(functor,&string)
    &&     TermIsInteger(arity,&value)
    &&     TermUnify(PrologArg(3),
               TermMakeCompound("/",2,functor,arity));
}

void Cboot(void)
{
    CPRIM("make_indicator",3,make_indicator_3);
}
```

See also

TermMakeFunctor(), TermMakeUniversal(), TermIsCompound()

Generate a Prolog floating-point number

```
#include "cpred.h"
```

```
TERM TermMakeFloat(double Value)
```

The function `TermMakeFloat()` generates a Prolog floating-point number and returns it as the result.

Arguments

Value	Floating-point number

Exceptions

The following error causes IF/Prolog to be exited:

implementation_error: TermMakeFloat : out of memory
: The function `TermMakeFloat()` could not be executed successfully due to a lack of memory.

See also

TermMakeInteger(), TermIsFloat()

Generate a Prolog structure

```
#include "cpred.h"
```

```
TERM TermMakeFunctor(STRING Functor, ARITY Arity)
```

The function `TermMakeFunctor()` generates a Prolog structure with the name *Functor* and the arity *Arity* and returns it as the result.

A new Prolog variable is generated for each argument of the structure.

Arguments

Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

implementation_error: TermMakeFunctor : out of memory
The function `TermMakeFunctor()` could not be executed successfully due to a lack of memory.

Example

The following simple C function realizes a predicate `make_struct/3`, which makes a structure with variables as arguments from given name and arity.

```
#include "cpred.h"

BOOLEAN make_struct_3(void)
/*
** make_struct( +Functor, +Arity, -Structure )
*/
{
    STRING string;
    long value;

    return TermIsAtom(PrologArg(1),&string)
    &&     TermIsInteger(PrologArg(2),&value)
```

```
    &&      TermUnify(PrologArg(3),
                TermMakeFunctor(string,value));
}

void Cboot(void)
{
    CPRIM("make_struct",3,make_struct_3);
}
```

See also

TermMakeCompound(), TermMakeUniversal(), TermIsFunctor()

Generate a Prolog integer

```
#include "cpred.h"
```

```
TERM TermMakeInteger(long Value)
```

The function `TermMakeInteger()` generates a Prolog integer and returns it as the result.

Arguments

Value	Integer

Exceptions

The following error causes IF/Prolog to be exited:

implementation_error: TermMakeInteger : out of memory
The function `TermMakeInteger()` could not be executed successfully due to a lack of memory.

See also

TermIsInteger()

Generate a Prolog list

```
#include "cpred.h"
```

```
TERM TermMakeList(TERM Head, TERM Tail)
```

The function `TermMakeList` generates a Prolog list where *Head* is used as the list head and *Tail* as the list tail. The generated list is returned as the result.

If *Head* or *Tail* is `NULL`, a Prolog variable is generated instead and is used as the list argument.

Arguments

Head	Term variable
Tail	Term variable

Exceptions

The following error causes IF/Prolog to be exited:

implementation_error: TermMakeList : out of memory
: The function `TermMakeList()` could not be executed successfully due to a lack of memory.

Example

The following simple C function realizes a predicate `mk_list/3`, which makes a list of integers up to a given limit.

```
#include "cpred.h"

BOOLEAN mk_list_2(void)
/*
** mk_list( +Count, -List )
*/
{
    long max, i;
    TERM list;

    list = TermMakeNil();
    if (!TermIsInteger(PrologArg(1),&max))
        return FALSE;
```

```
    for (i=max;i>=1;i--)
        list = TermMakeList(TermMakeInteger(i),list);

    return TermUnify(PrologArg(2),list);
}

void Cboot(void)
{
    CPRIM("mk_list",2,mk_list_2);
}
```

See also

TermMakeNil(), TermIsList()

Generate an empty list

```
#include "cpred.h"
```

```
TERM TermMakeNil(void)
```

The function `TermMakeNil()` generates an empty list and returns it as the result.

Example

See example at `TermMakeList()`.

See also

TermMakeList(), TermIsNil()

Generate a Prolog structure

```
#include "cpred.h"
```

```
TERM TermMakeUniversal(STRING Functor, ARITY Arity, TERM Args[])
```

The function `TermMakeUniversal()` generates a Prolog structure with the name *Functor*, the arity *Arity* and the arguments *Args* and returns it as the result.

If an element of the vector is `NULL`, a new Prolog variable is generated for this argument, and it is used as the structure argument.

Arguments

Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$
Args	Vector with *Arity* term variables

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

implementation_error: TermMakeUniversal : out of memory
The function `TermMakeUniversal()` could not be executed successfully due to a lack of memory.

Example

The following simple C function realizes a predicate `make_assignment/3`, which makes a term of the form *Term1*=*Term2* of its arguments:

```
#include "cpred.h"

BOOLEAN make_assignment_3(void)
/*
** make_assignment( +Term1, +Term2, -Assignment )
*/
{
    TERM terms[2];
```

```
    terms[0] = PrologArg(1);
    terms[1] = PrologArg(2);

    return TermUnify(PrologArg(3),
                TermMakeUniversal("=",2,terms));
}

void Cboot(void)
{
    CPRIM("make_assignment",3,make_assignment_3);
}
```

See also

TermMakeCompound(), TermMakeFunctor(), TermIsUniversal()

Generate a Prolog variable

```
#include "cpred.h"
```

```
TERM TermMakeVar(void)
```

The function `TermMakeVar()` generates a new Prolog variable and returns it as the result.

Exceptions

The following error causes IF/Prolog to be exited:

implementation_error: TermMakeVar : out of memory
: The function `TermMakeVar()` could not be executed successfully due to a lack of memory.

See also

TermIsVar()

Generate a Prolog list

```
#include "cpred.h"
```

```
TERM TermOpenList(TERM *Tail)
```

The function `TermOpenList()` is used to prepare the construction of a Prolog list.

Technically the function `TermOpenList()` generates a new Prolog variable and stores it in two term variables.

Tail is initialized with one of these term variables. The other term variable is returned as the result.

Arguments

Tail	Pointer to a term variable

Exceptions

The following error causes IF/Prolog to be exited:

implementation_error: TermOpenList : out of memory
: The function `TermOpenList()` could not be executed successfully due to a lack of memory.

Example

See example at `TermAddList()`.

See also

TermAddList(), TermCloseList()

Classify term

```
#include "cpred.h"
```

```
TERMTYPE TermType(TERM Term)
```

The function `TermType()` returns the type of the term *Term*.

The function returns the following values:

TERM_VAR
: The term is an uninstantiated Prolog variable.

TERM_ATOM
: The term is a Prolog atom.

TERM_INTEGER
: The term is an integer within the range of values for a C variable of the type `signed long`.

TERM_FLOAT
: The term is a floating-point number.

TERM_COMPOUND
: The term is a structure.

TERM_OTHER
: The term has a type not listed above.

Arguments

Term	Term variable

Hints

Future versions of IF/Prolog will extend the definition of the data type `TERMTYPE`. If the `TermType()` function is used, the user should make sure that the function can also return a value that is not currently defined as the result.

Example

The following simple C function realizes a predicate `type_of/2`, which gives the type to a given term.

```
#include "cpred.h"

BOOLEAN type_of_2(void)
/*
** type_of( +Term, -Type )
*/
{
    STRING typename;

    switch ( TermType(PrologArg(1)) )
    {
    case TERM_VAR:
        typename = "var";
        break;
    case TERM_INTEGER:
        typename = "integer";
        break;
    case TERM_FLOAT:
        typename = "float";
        break;
    case TERM_ATOM:
        typename = "atom";
        break;
    case TERM_COMPOUND:
        typename = "compound";
        break;
    default:
        typename = "unknown";
    }

    return TermUnifyAtom(PrologArg(2), typename);
}

void Cboot(void)
{
    CPRIM("type_of",2,type_of_2);
}
```

See also

TermDecompose()

Unify terms

```
#include "cpred.h"
```

```
BOOLEAN TermUnify(TERM Term1, TERM Term2)
```

The function `TermUnify()` unifies *Term1* with *Term2*.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term1, *Term2*	Term variables

See also

TermUnifyVar()

Unify term with atom

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyAtom(TERM Term, STRING Name)
```

The function `TermUnifyAtom()` unifies the term *Term* with an atom with the name *Name*. If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable
Name	String

Hints

The call of

```
TermUnifyAtom(Term,Name)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeAtom(Name))
```

See also

TermUnify(), TermMakeAtom()

Unify term with structure

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyCompound(TERM Term, STRING Functor,
ARITY Arity, ...)
```

The function **TermUnifyCompound()** unifies the term *Term* with a structure with the name *Functor* and the arity *Arity*, the arguments of which are the terms passed in ....

If one of the term variables passed in ... is `NULL`, a new Prolog variable is used instead.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable
Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$
...	*Arity* pointers to term variables or `NULL`

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

implementation_error: TermUnifyCompound : out of memory
The function `TermUnifyCompound()` could not be executed successfully due to a lack of memory.

Hints

The call of

```
TermUnifyCompound(Term,Functor,Arity,...)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeCompound(Functor,Arity,...))
```

Example

The following simple C function realizes a predicate `mk_indicator/3`, which makes a predicate indicator from given name and arity.

```
#include "cpred.h"

BOOLEAN mk_indicator_3(void)
/*
** mk_indicator( +Functor, +Arity, -Indicator )
*/
{
    STRING string;
    long value;
    TERM functor = PrologArg(1);
    TERM arity = PrologArg(2);

    return TermIsAtom(functor,&string)
    &&     TermIsInteger(arity,&value)
    &&     TermUnifyCompound(PrologArg(3),
               "/",2,functor,arity);
}

void Cboot(void)
{
    CPRIM("mk_indicator",3,mk_indicator_3);
}
```

See also

TermUnifyFunctor(), TermUnifyUniversal(), TermUnify(), TermMakeCompound()

Unify with floating-point number

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyFloat(TERM Term, double Value)
```

The function `TermUnifyFloat()` unifies the term *Term* with a term that corresponds to the floating-point number *Value*.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable
Value	Floating-point number

Hints

The call of

```
TermUnifyFloat(Term,Wert)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeFloat(Wert))
```

See also

TermUnify(), TermMakeFloat()

Unify term with structure

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyFunctor(TERM Term, STRING Functor, ARITY Arity)
```

The function `TermUnifyFunctor()` unifies the term *Term* with a structure with the name *Functor* and the arity *Arity*, the arguments of which are uninstantiated variables.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable
Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

implementation_error: TermUnifyFunctor : out of memory
The function `TermUnifyFunctor()` could not be executed successfully due to a lack of memory.

Hints

The call of

```
TermUnifyFunctor(Term,Functor,Arity)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeFunctor(Functor,Arity))
```

Example

The following simple C function realizes a predicate `mk_struct/3`, which makes a structure with variables as arguments from given name and arity.

```
#include "cpred.h"

BOOLEAN mk_struct_3(void)
/*
** mk_struct( +Functor, +Arity, -Structure )
*/
{
    STRING string;
    long value;

    return TermIsAtom(PrologArg(1),&string)
    &&     TermIsInteger(PrologArg(2),&value)
    &&     TermUnifyFunctor(PrologArg(3),string,value);
}

void Cboot(void)
{
    CPRIM("mk_struct",3,mk_struct_3);
}
```

See also

TermUnifyCompound(), TermUnifyUniversal(), TermUnify(), TermMakeFunctor()

Unify term with integer

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyInteger(TERM Term, long Value)
```

The function `TermUnifyInteger()` unifies the term *Term* with a term corresponding to the integer *Value*.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable
Value	Number with which the term is to be unified

Hints

The call of

```
TermUnifyInteger(Term,Wert)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeInteger(Wert))
```

See also

TermUnify(), TermMakeInteger()

Unify term with list

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyList(TERM Term, TERM Head, TERM Tail)
```

The function `TermUnifyList()` unifies the term *Term* with a list whose list head is *Head* and whose tail is *Tail*.

If *Head* or *Tail* is `NULL`, a new Prolog variable is used instead.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable
Head, *Tail*	Term variable or `NULL`

Hints

The call of

```
TermUnifyList(Term,Head,Tail)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeList(Head,Tail))
```

See also

TermUnify(), TermMakeList()

Unify term with the empty list

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyNil(TERM Term)
```

The function `TermUnifyNil()` unifies the term *Term* with the empty list.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable

Hints

The call of

```
TermUnifyNil(Term)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeNil())
```

See also

TermUnify(), TermMakeNil()

Unify term with structure

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyUniversal(TERM Term, STRING Functor, ARITY
Arity, TERM Args[])
```

The function `TermUnifyUniversal()` unifies the term *Term* with the structure with the name *Functor* and the arity *Arity* and the arguments *Args*.

If a term in *Args* is `NULL`, a new Prolog variable is used instead.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term	Term variable.
Functor	Character string, name of the structure
Arity	Integer, $0 \leq Arity \leq 127$
Args	Vector with *Arity* term variables/`NULL`

Exceptions

The following errors cause IF/Prolog to be exited:

implementation_error: *Arity* **: arity out of range**
The argument *Arity* is not within the range 0..127.

implementation_error: TermUnifyUniversal : out of memory
The function `TermUnifyUniversal()` could not be executed successfully due to a lack of memory.

Hints

The call of

```
TermUnifyUniversal(Term,Functor,Arity,Args)
```

is, except for optimization, identical to

```
TermUnify(Term,TermMakeUniversal(Functor,Arity,Args))
```

Example

The following simple C function realizes a predicate `mk_assignment/3`, which makes a term of the form *Term1=Term2* of two given arguments.

```
#include "cpred.h"

BOOLEAN mk_assignment_3(void)
/*
** mk_assignment( +Term1, +Term2, -Assignment )
*/
{
    TERM terms[2];

    terms[0] = PrologArg(1);
    terms[1] = PrologArg(2);

    return TermUnifyUniversal(PrologArg(3),"=",2,terms);
}

void Cboot(void)
{
    CPRIM("mk_assignment",3,mk_assignment_3);
}
```

See also

TermUnifyCompound(), TermUnifyFunctor(), TermUnify(), TermMakeUniversal()

Unify terms

```
#include "cpred.h"
```

```
BOOLEAN TermUnifyVar(TERM Term1, TERM Term2)
```

The function `TermUnifyVar()` unifies *Term1* with *Term2*.

If unification is successful, the function returns `TRUE`, otherwise it returns `FALSE`.

Arguments

Term1, *Term2*	Term variables

Hints

This function is defined for reasons of orthogonality.

The call of

```
TermUnifyVar(Term1,Term2)
```

is identical to

```
TermUnify(Term1,Term2)
```

See also

TermUnify()

Chapter 9

Prolog syntax

Starting from the basic Prolog character set, the syntax of Prolog terms is built up step by step in the sections *Basic Prolog elements* and *Terms*. The remaining parts of this chapter deal with special aspects of operators, arithmetic expressions and functions, comparison predicates and list processing.

9.1 Basic Prolog elements

The character set used by Prolog is divided into four classes of characters, the Prolog-specific characters being of particular importance in the Prolog syntax. The remaining characters form lexical elements and delimiters.

Prolog character set

The **Prolog character set** consists of the following classes of characters:

- Letters, digits, underscore character
- Special characters
- Prolog-specific characters
- Delimiters

These classes comprise the following individual characters:

Letters, digits, underscore character

```
a   b   c   d   ...   z
A   B   C   D   ...   Z
0   1   2   3   4   5   6   7   8   9   _
```

Special characters

```
#    &    *    +    -    .    /    :    ;
?    @    /    ^    $         <    =    >
```

Prolog-specific characters

```
,    !    (    )    {    }    [    ]    |    '    "    %
```

Delimiters

```
Blank, Tab character, New line character
```

Prolog-specific characters

The Prolog-specific characters have a special meaning in the Prolog syntax:

Character	Meaning
,	Separates arguments in structures, elements in lists and conjuncted subgoals.
!	Cut
( )	Part of term and clause syntax. Parentheses must always occur in pairs. If one of the two parentheses is missing, Prolog will report a syntax violation.
[ ]	Part of term syntax (list notation). Square brackets must always occur in pairs. If one of the two square brackets is missing, Prolog will report a syntax violation.
{ }	Enclose subconditions in grammar rules. Braces must always occur in pairs. If one of the two braces is missing, Prolog will report a syntax violation.
"	A string of characters enclosed in double quotes is interpreted according to the value of the Prolog flag `double_quotes`. Double quotes must always occur in pairs.
'	A string of characters enclosed in single quotes is interpreted as an atom. Single quotes must always occur in pairs.
|	List tail delimiter: delimits the list tail from leading list elements.
.	End of a term in a Prolog input, list functor or decimal point for floating-point numbers.
%	Start of a comment which is terminated by end-of-line.
/* */	Identify the beginning and end of a comment.

Lexical elements and delimiters in Prolog

In inputs, Prolog makes a distinction between lexical elements and delimiters.

Lexical elements
When Prolog text is read (with the predicate `read/1`, consulting or compiling), syntactical analysis is performed. The internal representation is built up from the lexical elements (**tokens**) of the Prolog language, namely **atoms**, **variables**, **integers** in different notations, **floating-point numbers**, **rational numbers** and the **Prolog-specific characters**.

Atoms are character sequences which are used, for example, as names. The following are valid atoms:

- character sequences beginning with a lowercase letter and consisting only of letters, digits and underscores;
- character sequences consisting of special characters (e.g. `?-`) and
- character strings enclosed in single quotes.

Variables are also sequences of letters, digits and underscores which begin with an uppercase letter or with an underscore `_` to distinguish them from atoms.

Integers are sequences of digits, possibly preceded by the minus sign (`-`).

Floating-point numbers are sequences of digits containing a decimal point or an exponent, possibly preceded by the minus sign.

Octal numbers consist of a prefix `0o` (Zero-o) for octal, and a sequence of octal digits (from `0` to `7`), the whole possibly preceded by the minus sign.

Hexadecimal numbers consist of a prefix `0x` (Zero-x) for hexa, and a sequence of hexadecimal digits (from `0` to `9`, from `a` to `f` and from `A` to `F`), the whole possibly preceded by the minus sign.

Rational numbers (extension) consist of a prefix `0r` (Zero-r), and a ratio numerator/-denominator, the whole possibly preceded by the minus sign.

Bracket pairs `( ) [ ]` are used to denote structures and lists.

Character strings enclosed in double quotes `"` are interpreted according to the value of the Prolog flag `double_quotes`. The interpretation is either a list of character codes, a list of characters or an atom.

Separators
The Prolog system must be able to recognize the various lexical elements in Prolog during input, in one of two ways:

- The beginning of a new lexical element must be recognizable.

  Example:

  ```
  female(anna)
  ```

  The opening parenthesis cannot belong to the token `female`

- The individual elements must be separated from each other by means of explicit delimiters.

  Example:

  ```
  Xis5
  ```

  Unless additional delimiters are specified Prolog will interpret this sequence as a single lexical element.

Separators in Prolog are blanks, the end-of-line character ↩ , the tab character and comments.

Comments in Prolog may be one of two kinds:

- variable-length comments
- end-of-line comments

The Prolog system registers comments merely as delimiters between lexical elements. To the user they also serve for documentation purposes.

Variable-length comments

All characters from the pair `/*` to the occurrence of the pair `*/` are interpreted as a comment.

The comment may extend over more than one line. Nested comments are prohibited, if the Prolog flag `nested_comments` has the value `off` and allowed if the flag has the value `on`.

Comments may be included between lexical elements in Prolog texts wherever delimiters are permitted. Thus, a comment may even be inserted in Prolog normal structures or in lists.

```
X = illness( /* 25% probability */ flu).
```

End-of-line comments

An end-of-line comment starts with the percent sign (%). All characters between the percent sign and the end of the line are then interpreted as a comment.

Delimiters in normal structures and expressions

- In normal structures delimiters are prohibited between the functor and the opening parenthesis.
- A delimiter must occur between an operator and a subsequent expression enclosed in parentheses. If it is omitted, Prolog will interpret the operator together with the expression in parenthesis as a normal structure.

  X is 5+↩
  (2*3).

Blanks not functioning as delimiters
In strings delimited by single or double quotes, blanks are part of the string and thus do not function as delimiters.

Disabling the ↩
The return key ↩ retains its delimiter function even in strings enclosed in single or double quotes, unless it is disabled by the backslash. The disabled end-of-line character is not included in atoms or character lists. Without the preceding backslash, ↩ will result in a syntax violation if used in these strings.

```
'abcde\
fgh'
```

corresponds to

```
'abcdefgh'
```

Making inputs easier to read
Delimiters can be used to make Prolog text easier to read. Delimiters (tabs, blanks, comments, new line characters) may be inserted between any lexical elements, except between the functor and opening parenthesis in normal structures, as otherwise the functor would be interpreted as an operator.

Examples of dividing into lexical elements

Prolog can recognize lexical elements in the following cases, among others:

`3is 1+2.`
: The characters `i` and `+` each indicate the end of the preceding number; blanks are not required. However, there must be at least one blank between `is` and `1`, as Prolog would otherwise interpret the input as the atom `is1`.

`X=[].`
: Blanks are not required.

`10/ (2+3)`
: The slash indicates the end of the preceding number. There must be at least one blank between the slash and the opening parenthesis, as Prolog would otherwise interpret the input as a normal structure with the functor `/`

`10/'+'(2,3)`
: Blanks are not required, as the operator `'+'` is separated from the preceding operator `/` by the single quotes.

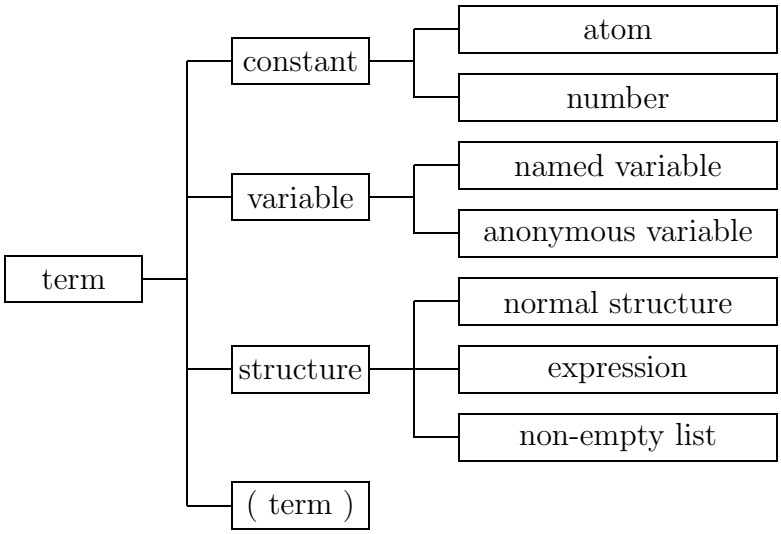
`a'likes'b.`
: Blanks are not required (with the delimiting single quotes, `'likes'` is unambiguously identifiable as an atom; `'likes'` must be defined as an infix operator).
If the single quotes are omitted, delimiters would be required for separation: `a likes b.`

9.2 Terms

Term is the generic for all syntactic units in Prolog.

A term may belong to one of the following classes:

- constant
- variable
- structure
- term enclosed in parentheses.



Test predicates

Prolog provides a number of built-in predicates that enable you to check what class a term belongs to.

Predicate	**tests whether a term is ...**
atomic/1	a constant (number or atom)
atom/1	an atom
letter/1	an atom consisting of one letter
digit/1	an atom consisting of one digit
number/1	a number
integer/1	an integer number
float/1	a floating-point number
rational/1	a rational number (extension)
var/1	an uninstantiated variable
nonvar/1	not an uninstantiated variable
compound/1	a structure

When a term is read any superfluous parentheses are ignored, so the query

```
[user] ?- atomic( (( a )) ).

yes
```

succeeds because `a` is an atom.

9.2.1 Constants

In Prolog there are two different kinds of **constants**:

- atoms
- numbers

Atoms

Atoms are defined as follows:

- character strings consisting of letters, digits and underscore characters and beginning with a lowercase letter;
- any character strings enclosed in single quotes. Even the "empty" character string - two single quotes - is an atom. Each single quote within an atom must be represented by two single quotes or be disabled by means of a preceding backslash (\');
- character strings made up of special characters;
- the empty list, represented by a pair of square brackets (`[]`), is also interpreted as an atom.

Special requirements for entering atoms

The backslash has a special meaning when appearing in atoms (the atoms must then be enclosed in single quotes):

- If the backslash is followed by one of the following letters, this combination is interpreted as a control character:

String	ASCII (decimal)	Meaning
\a	7	Bell (Alert)
\b	8	Backspace
\t	9	Horizontal tab
\n	10	Line feed
\v	11	Vertical tab
\f	12	Form feed
\r	13	Carriage return
\e	27	Escape (ESC)

- If the backslash is followed by new line (key ↩), the new line character ignored, so that long atoms can be entered easily.

  'abc\↩
  def'

  equivalent to

  ```
  abcdef
  ```

- If the backslash is followed by an apostrophe (’), the delimiter function of the apostrophe is disabled: \’ represents the apostrophe character (\’ has the same effect as a double apostrophe ’’).

  ```
  'abc\'de''fg'
  ```

  equivalent to

  ```
  abc'de'fg
  ```

- To represent the printable character \ you have to disable the backslash by entering it twice:

  ```
  'h\\a'
  ```

  equivalent to

  ```
  h\a
  ```

- If the backslash is followed by up to three octal digits (0...7) and a further backslash, this sequence is equivalent to the character with this character code. The character code must be in the range 1 to 255 (octal 377). In this way nonprintable characters can be represented.

  ```
  'abc\040\def'
  ```

  equivalent to

  ```
  'abc def'
  ```

- If the backslash is followed by `x`, two hexadecimal characters (`0`...`9`, `a`...`f`, `A`...`F`) and a further backslash, this sequence is equivalent to the character with this character code. The character code must be in the range 1 to 255. In this way nonprintable characters can be represented.

  ```
  'abc\x64\ef'
  ```

  equivalent to

  ```
  'abcdef'
  ```

In all other cases, the syntax violation `illegal escape sequence` is reported.

Examples of atoms

```
anna
m5
has_account_number
append
'1'
'Nomen est omen.'
'Don''t forget that single quotes must be duplicated'
'It\'s also possible to escape them'
'\n line feed before new text'
!
=:=
#?>=<*
```

The length of an atom (number of characters) may be queried with the predicate `atom_length/2`.

Numbers

Numbers in Prolog may be either (binary, octal, decimal, hexadecimal) integers, rationals, or floating-point numbers.

Integers are sequences of (binary, octal, decimal, hexadecimal) digits, optionally preceded by a minus sign (`-`). An ASCII character preceded by the prefix Zero-quote (0') denotes an integer, which represents its ASCII code.

$$\left[\begin{array}{c} \\ - \\ \\ \end{array}\right] \left\{\begin{array}{l} \textit{sequence of digits} \\ 0x\ \textit{hexadecimal sequence} \\ 0o\ \textit{octal sequence} \\ 0b\ \textit{binary sequence} \\ 0'\ \textit{ASCII character} \end{array}\right\}$$

Sequence of digits

Sequence of digits (0...9)

Hexadecimal sequence

sequence of hexadecimal characters (0...9, *a...f*, *A...F*)

Octal sequence

Sequence of octal characters (0...7)

Binary sequence

Sequence of binary characters (0, 1)

ASCII character

a single character (*see Atoms*)

The full machine word range (32 or 64 bits) is available for integer arithmetic. The following applies:

`minint` $\leq$ N $\leq$ `maxint`

The values for minint and maxint can be accessed by means of the zero arity functions `minint/0` and `maxint/0`, e.g. using the query

```
[user] ?- X is minint.
```

If the Prolog system has been configured with multiple precision integer support, you can use integers outside this number range.

Character codes are also integers:

```
[user] ?- X is 0'a + 1, char_code(C, X).

X       = 98
C       = b

yes

[user] ?- X is 0'\n.

X       = 10

yes
```

Examples of integers

```
1256
-000000034
0xa34f       % This is a hexadecimal number
0o1277       % This is an octal number
0b1010       % This is a binary number
0'a          % Character codes are integers
0'\n
```

Floating-point numbers are sequences of decimal digits containing a decimal point or an exponent or both, optionally preceded by a minus sign `-`.

$$\left[\begin{array}{c}\\ - \end{array}\right]\left\{\begin{array}{l} N1.N2 \\ \left\{\begin{array}{l} N1 \\ N1.N2 \end{array}\right\}\left\{\begin{array}{l} E \\ e \end{array}\right\}\left[\left\{\begin{array}{l} + \\ - \end{array}\right\}\right] exponent \end{array}\right\}$$

$N1$, $N2$, *exponent* are sequence of digits.

If you enter a floating-point number outside the value range, it will be set to the nearest representable value.

Floating-point format
By using the predicate `float_format/2` you can control the format which the Prolog system uses to output floating-point numbers.

Examples of floating-point numbers

```
22.99
-1.0
123.45E10
-21e-20
```

Rational numbers are marked by a predeeding `0r`.

[−] 0*r sequence of digits / sequence of digits*

Examples of rational numbers

```
-0r1/3
0r1256/1000000000
```

9.2.2 Variables

In Prolog there are two different kinds of variables:

- Prolog variables
  These clearly differ in concept from variables in conventional programming languages. Since this type of variable is the one most commonly used, it will often be referred to simply as "variable".
- Global variables
  These are identical in concept to variables in conventional programming languages such as C or Fortran.

Prolog variables

A Prolog variable can be instantiated to any term, regardless of whether the term is a constant, a variable or a structure. During goal execution, Prolog variables are instantiated to terms by unification and therefore assume all the attributes of these terms. This instantiation remains in force during goal execution and is not undone until backtracking takes place. Any attempt to instantiate an already instantiated variable to another value will lead to the failure of the unification and will initiate backtracking to the last choice point. If two variables are instantiated to each other, they will be regarded as identical in subsequent processing.

```
[user] ?- var(X), X = a, nonvar(X), atom(X).

X       = a

yes

[user] ?- X = a, var(X).

no
```

```
[user] ?- X \== Y, X = Y, X == Y.

X       = _100
Y       = _100

yes
```

If a toplevel goal contains variables, the associated variable instantiations, if any, will be output if the query is successful. A variable instantiation made during a program run will remain in force and cannot be overwritten until backtracking takes place, which backtracks before the goal which instantiated the variable.
This means, for example, that queries such as

```
[user] ?- N = 5, N is N + 1.
```

will always lead to a failure in Prolog (`N` is instantiated to the constant `5` and `5 is 5 + 1` cannot be satisfied).
To avoid this failure you have to use an additional variable which is instantiated to the result of the addition operation:

```
[user] ?- N = 5, M is N + 1.

N       = 5
M       = 6

yes
```

Variable names

Variables are identified syntactically by a name.

The **variable name** is a character sequence made up of letters, digits and underscore characters and beginning with an uppercase letter or underscore character.

On its own, an underscore character (`_`) stands for an anonymous variable.

Scope of a variable

The scope of a variable is a clause; within this clause, variables with the same name are identical. Variables having the same name but occurring in different clauses are different variables.

Anonymous variable

In Prolog, a special role is played by the "anonymous variable", which is indicated syntactically by _ (underscore character).

Each occurrence of _ corresponds to a different variable; even within a clause, _ does not stand for one and the same object. Wherever a variable is used only once within a clause, the anonymous variable can and should be used to emphasize this fact.

If you use an anonymous variable in a toplevel Prolog query, no variable instantiation will be output for the anonymous variable.

Single-variable-check

Prolog assumes that a variable with a name beginning with a character other than _ will be used to create relationships within a clause and must therefore be used more than once. An appropriate check is carried out when the Prolog program is read in. The occurrence of variables which are used only once is reported with the message `warning: single_variable`. This is a good way of detecting typing errors which would otherwise affect the behavior of Prolog programs.

This check is not carried out for variables which begin with an underscore. You can use this fact, for example, to make programs easier to read.

```
fact(_NO,_SINGLE,_VAR,_CHECK).
```

Examples for variable

```
X
Key_8
License_plate
_ancestor
BOSS
_
H1
```

Global variables

The global variable concept differs considerably from that of the usual Prolog variables, being derived more from the variable concept of procedural programming languages such as C and Fortran. Global variables are assigned values not by unification, but by calling special predicates (`set_global/2`, `push_global/2`).

Name

The name of a global variable is an atom.

Definition

Global variables are defined with `set_global/2` or `push_global/2`, at which time they are assigned a value. The names of the global variables declared in the module can be queried with `current_global/1`. The definition of a global variable is canceled with the predicate `unset_global/1`.

Value

Any term is valid as the value for a global variable. The creation and removal of global variables and the assignment of values to them are side effects of Prolog goal execution and are therefore not affected by backtracking. In particular, the value assignment can be overwritten as often as you like by calling `set_global/2`.

Temporal validity

A global variable is valid from the moment it is defined by means of `set_global/2` or `push_global/2`.
It is retained for the remainder of the Prolog session even after a procedure has been terminated.
A global variable loses its validity when the goal `unset_global/1` is called or at the end of the Prolog session.

Logical scope

In contrast to Prolog variables, global variables can be accessed from all predicates by their names. Global variables are set up on a module-specific basis and are visible only in this module. All the predicates which are used to process global variables are metapredicates, so that it is possible to access a global variable in a different module by specifying the appropriate module name.

```
[user] ?- set_global(var, 5) @ my_module.

yes

[user] ?- current_global(var).

no

[user] ?- current_global(var) @ my_module.

yes
```

With global variables you can exchange information between different predicates without having to pass this information using explicitly specified arguments.

Stacks of global variables

A stack of values can be administered under the name of a global variable. The following rules apply:

- `set_global/2` creates a stack with precisely one entry, and at the same time any existing stack for the same variable is deleted.
- `push_global/2` adds a further entry to an existing stack as the top element. If the global variable does not yet exist, a stack with one entry is created.
- `get_global/2` is used to query the value of the top element. If the global variable does not exist, the Prolog system reports an error.
- `pop_global/2` is used to query the value of the top element. The top value is then removed from the stack. When the last entry in the stack is removed, the global variable is also deleted. If the global variable does not exist, the Prolog system reports an error.
- `unset_global/1` removes the entire stack and the global variable.

Example of global variables

```
[user] ?- set_global(glob, 1), get_global(glob, Pro1),
write(Pro1), nl,
set_global(glob, nun), get_global(glob, Pro2),
write(Pro2), nl, write(glob).
1
nun
glob
Pro1 = 1
Pro2 = nun

yes
```

In this example the global variable `glob` is assigned a value more than once during goal execution, and the old value is overwritten. This would not be possible with Prolog variables.

9.2.3 Structures

Unlike constants and variables, structures consist of more than one component. These individual components, in turn, may be any terms, i.e. they themselves may also be structures.

In Prolog, structures can be represented syntactically in different ways.
A structure can be represented as one of the following:

- normal structure
- expression
- non-empty list

Every structure can be represented as a normal structure, which is therefore a general way of representing structures.
Expressions and lists are special forms which simplify the representation of structures. For every expression and every non-empty list there is an equivalent normal structure.

Normal structures

A **normal structure** is defined as follows:

$$functor(\ Argument_1, \ldots,\ Argument_n\ )$$

functor must be an atom; $Argument_n$ ($1 \leq n \leq$ `max_arity`) may be any term. The number of arguments is referred to as **arity**. The maximum arity of a structure can be queried with the predicate `current_prolog_flag/2`:

```
[user] ?- current_prolog_flag(max_arity, M).

M       = 127

yes
```

Note

Some built-in predicates interpret atoms as structures with zero arity.
However, the following applies:

```
[user] ?- compound(atom).

no
```

Examples of normal structures

```
parents(father, mother)
faculty(10, Result)
date(year(1984), month(X), day(holiday(2, usa)))
+(5, 6)
*(5, +(2,9))
@(:(user, likes( jones, tiger),), user)
        % == user:jones likes tiger @ user
```

Expressions

Expressions are structures which have an operator as a functor and are specified in operator notation.

Operator notation is used to specify terms in mathematical notation or to make predicates clearer and easier to read. For example, `jones likes tiger` is easier to read than `likes(jones,tiger)` and `1 + 2` more familiar than `+(1, 2)`. In particular, mathematical expressions need not be reformulated prior to input into the unfamiliar normal structure notation.

Depending on whether the operator is declared as an infix, prefix or postfix operator, an expression may have the following form:

$$\left\{ \begin{array}{l} argument_1 \;\; infix_operator \;\; argument_2 \\ prefix_operator \;\; argument_1 \\ argument_1 \;\; postfix_operator \end{array} \right\}$$

$argument_1$, $argument_2$ are any terms.

`infix_operator` is an operator declared with `xfx`, `xfy` or `yfx`.

`prefix_operator` is an operator declared with `fx` or `fy`.

`postfix_operator` is an operator declared with `xf` or `yf`.

A number of common operators are declared automatically by the Prolog system. A list of these predefined operators can be found in the section 9.3.2 (*List of built-in operators*). Using `op/3`, further operators can be declared and canceled again. `op/3` can also be used to cancel any built-in operator declaration.

At least one delimiter must come between an operator and a subsequent term in parentheses. Otherwise Prolog will interpret the operator as the functor of a normal structure.

For internal processing, expressions in operator notation are converted to the equivalent normal structure.

When terms are output with a `write` or `writeq` predicate, expressions are shown in operator notation; the same applies to when clauses are listed using `listing/0/1`.
With the predicate `write_term/2/3`, terms can be output in normal structure notation if the option `ignore_ops(true)` is specified.

```
[user] ?- write_term( 3 is 2 + 1, [ignore_ops(true)]).
is(3,+(2,1))

yes
```

Examples of expressions

Many frequently used operators are built in. The following are therefore equivalent:

Operator notation	Equivalent normal structure
2 + 4 * 5	+(2, *(4, 5))
25 - 5 / (3 - 1)	-(25, /(5, -(3, 1)))
21 - *(2, 3)	-(21, *(2, 3))
\+ Goal	\+(Goal)
listing(likes / 2) @ my_module	@(listing(/(likes, 2)), my_module)

Once the following operators have been declared

```
[user] ?- op(200,yfx,likes).

yes

[user] ?- op(180,xf,is_a_god).

yes
```

the following expressions are equivalent:

Operator notation	Equivalent normal structure
jones likes tiger	likes(jones, tiger)
zeus is_a_god	is_a_god(zeus)

Lists

A `list` may be either of the following

$$\left\{ \begin{array}{l} empty\ \ list \\ non-empty\ \ list \end{array} \right\}$$

The **empty list** is represented as a pair of square brackets. `[]` is not a structure but an atom with a special meaning.

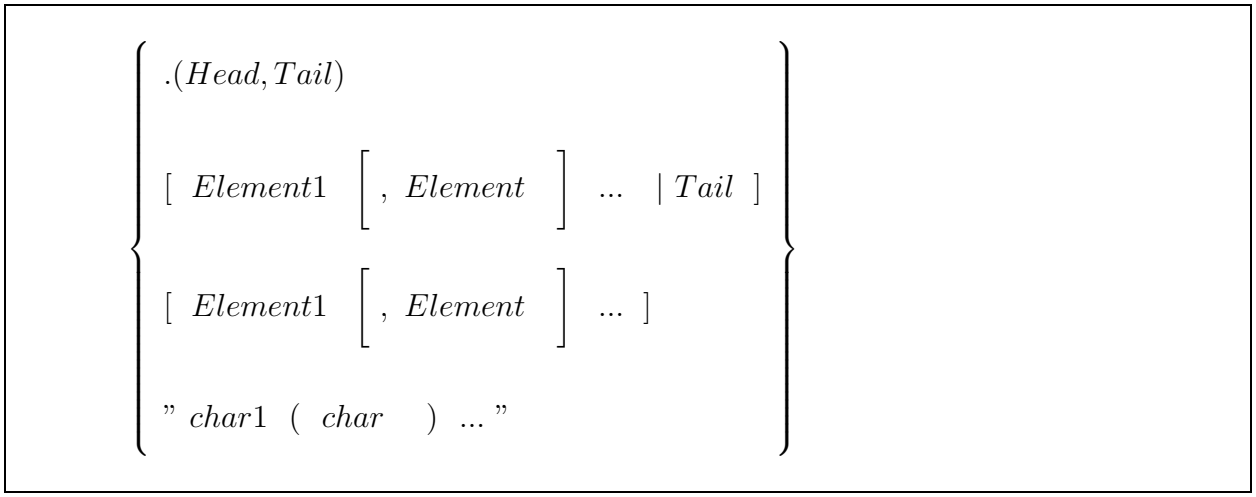
A **non-empty list** is equivalent to a normal structure of the form
`.(` *Head* `,` *Tail* `)`, where '`.`' is the list functor, *Head* indicates the first element in the list, and *Tail* indicates the list with the remaining elements, or the empty list.

To simplify the representation of lists, Prolog provides a special list notation; the elements are enclosed in square brackets, separated by commas: `[Element1, Element2, ... ]`.

Because splitting a list into a head and a tail is a common operation, a special notation has been provided for this purpose:
`[ Head | Tail ]`.
It is also possible to enter more than one element, separated by commas, in front of the

vertical bar that precedes the tail.
A string enclosed in double quotes (") is interpreted according to the value of the Prolog flag `double_quotes`. Each occurrence of double quotes inside the string must either be entered twice or be disabled by means of a preceding backslash.

The following notations are therefore possible for a non-empty list:



Element1, *Element*, *Head* may be any terms. *Tail* is a (possibly empty) list.

List-like structures

List-like structures can be generated by specifying a single element instead of a tail. These structures do not have all the characteristics inherent in lists.

```
[user] ?- append( .(a,b), [c], Y).

no

[user] ?- member( M, [a, b| c]).

M       = a;
M       = b;

no
```

It is not advisable to use such structures since they cannot be processed correctly by all predicates.

Converting structures into lists

The predicate `=../2` (univ) enables you to convert structures into non-empty lists and vice versa.

```
[user] ?- X =.. [write, user_output, abcdefgh], call( X).
abcdefgh
X       = write(user_output,abcdefgh)

yes
```

Examples of lists:

List	Equivalent normal structure
[a,b | [c,d]]	.(a , .(b, .(c, .(d,[ ]))))
[a | .(b , [c,d])]	.(a , .(b, .(c, .(d,[ ]))))
[[a]B]	.(.(a,[ ]),B)

9.3 Operators

Operators are atoms which were defined as operators by means of `op/3`. At the same time, the operator type, the operator precedence and the associativity of its arguments are defined. The following section describes how the conversion of an expression to the equivalent normal structure is controlled by these operator attributes. The section 9.3.2 (*List of built-in operators*) provides an overview of the built-in operators of the Prolog system. The section *Test and definition predicates* on page 820 deals with points to be taken into account in the operator definition.

9.3.1 Converting expressions into equivalent normal structures

At declaration time, the following attributes are defined for an operator:

- type
- operator precedence
- associativity of arguments.

These attributes determine the normal structure to which an expression in operator notation corresponds.

Operator type

In Prolog there are three types of operator:

- prefix operators (the operator is located in front of the argument)
- infix operators (the operator is located between two arguments)

- postfix operators (the operator is located after the argument)

If the arguments of an operator are themselves not expressions in operator notation, the type declaration is sufficient to determine its associated normal structure:

Expression	**Associated normal structure**
operator argument	operator(argument)
argument1 operator argument2	operator(argument1,argument2)
argument operator	operator(argument)

If an expression contains more than one operator, the type specification is not sufficient alone to uniquely determine which operator is the functor of the associated normal structure and how the arguments are assigned to the different operators. This assignment can be made unambiguous by means of parentheses.
To make the assignment unique even without parentheses, each operator is given the attributes 'operator precedence' and 'associativity of arguments'. These two attributes define parentheses implicitly.

Operator precedence

The precedence of an operator is represented by an integer between 1 and 1200.
It is defined in the operator declaration.

If an expression contains only operators with different precedence, the operator with the highest precedence is the main functor of the associated normal structure.

Expression	**Associated normal structure**
`1 + 2 * 3 ** 7`	`+( 1, *( 2, **( 3, 7)))`
`a :- b , c ; d`	`:-( a, ;( ','( b, c), d))`

In the second example, the comma acting as a functor of a normal structure must be specified in single quotes (`','`), as the Prolog system will always interpret a comma on its own in this context as a delimiter between the arguments of a structure.

Associativity of arguments

In order to ensure that the association between an expression and the corresponding normal structure is also unique in more general cases, the precedence of the terms involved and the precedence for binding (associativity) of the arguments of an operator are also required.

The precedence of terms is governed by the following rules:

- atoms (except for operators), numbers, variables, normal structures and lists have the precedence 0,

- terms enclosed in parentheses - (`term`) - have the precedence 0,
- expressions not enclosed in parentheses take the precedence of their associated operator.

Besides operator precedence, the operator declaration also defines the associativity of the arguments.
The overview below shows which options Prolog offers.
`f` stands for the operator, `x` indicates the position where only an argument with a precedence less than that of `f` is allowed to occur, and `y` indicates the position where the precedence of the argument must be less than or equal to that of `f`.

Notation	**operator Type**
fx	prefix operator
fy	prefix operator
xf	postfix operator
yf	postfix operator
xfx	infix operator
xfy	infix operator
yfx	infix operator

Precedence(x) < Precedence(f), Precedence(y) ≤ Precedence(f).

Operators declared with `fy` or `xfy` are called right-associative, and those declared with `yf` or `yfx` are called left-associative (see [11]).

To avoid ambiguity when converting parenthesized expressions with two or more operators, the operators you define yourself should either be assigned different precedence numbers or be declared with identical argument precedence attributes if the operator precedence is identical (i.e. operators with identical precedence should be either all left-associative or all right-associative).

Ambiguities may arise if you fail to do this.

Examples of left and right-associative operators:

```
[user] ?- op(500, xfy, plus).      % right-associative

yes

[user] ?- op(500, yfx, minus).     % left-associative

yes

[user] ?- write_term(1 plus 2 plus 3, [ignore_ops(true)]).
plus(1,plus(2,3))
yes

[user] ?- write_term(1 minus 2 minus 3, [ignore_ops(true)]).
```

```
minus(minus(1,2),3)
yes
```

Converting expressions with two or more operators into normal structures

The conversion of expressions is governed by the following rules:

- First, the highest precedence number (HP) of the operators in the expression is determined (= max(Precedence(f1), ... , Precedence(fn))).
- If the expression contains only one operator with Precedence(f) = HP, then this operator is the functor of the associated normal structure.
- If an expression contains two or more operators with Precedence(f) = HP, then the operator types defined for the arguments are used to determine which of these operators is the functor of the associated normal structure.
- Operands which themselves consist of expressions in parenthesis are evaluated separately.

Depending on the type of operator involved, this determines whether the associated normal structure has an arity of 1 or 2; no other arities are possible.
Arguments which are in turn expressions are converted into normal structures in analogous fashion.

```
2 - 5 + 2 * 7 / 2
```

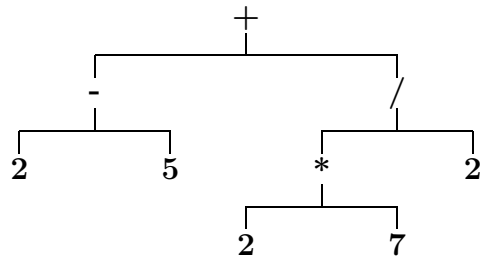
The following operator definitions apply:

```
op(500,yfx,+),
op(500,yfx,-),
op(400,yfx,*),
op(400,yfx,/).
```

The highest precedence number is `500`. Since the expression contains two operators with precedence `500`, the argument precedence definitions must also be taken into account in order to determine the main functor.
For the main functor `-`, the result would be `f = -`, the argument in position `y` would be `2` (precedence `0`), the argument in position `x` would be the expression `5 + 2 * 7 / 2`. This latter expression contains an operator with a precedence number of `500` and thus itself has precedence `500`, thereby contradicting the operator declaration `yfx`.
For `+`, the result is `f = +`; the argument in position `y` is `2 - 5` and the argument in position `x` is `2 * 7 / 2`, so that precedence(`x`) < precedence(`f`). As a result, `+` is the main functor of the associated normal structure.

Similarly, the argument `2 * 7 / 2` yields `/` as the functor of the associated normal structure.

This "splitting process" can be represented in a tree diagram as follows: The associated normal structure can be immediately read from the tree:



The left-hand "branch" represents the first argument of the normal structure, and the right-hand "branch" the second argument. Both arguments are in turn structures. The following normal structure is produced:

```
+( -(2,5) , /( *(2,7), 2) )
```

If two different operators have the same name, then one of these operators must be a prefix operator and the other an infix or postfix operator. All other combinations of different operators with the same name are invalid.

Operators are administered in modules. The operations of creating, redefining and canceling operators only ever act on the module in which the corresponding operator action was performed.

Operators are syntactical constructs which are taken into account when terms are read or written. If a structure is represented in operator notation instead of term notation, some structures will have different texts but will be identical in their internal representation. Assuming the following operator definitions:

```
:- op(100, fy, [say,die]).
:- op(200, yf, [say,die]).
```

the following terms will have an identical internal representation:

```
say never die
(never say) die
die say never
die (never say)
say(never) die
die say(never)
die(say never)
die(say(never))
```

The internal representation of a term can best be checked by outputting a term with the predicate `write_canonical/1/2` since this predicate ignores operator declarations.

If a name is defined both as a prefix and as a postfix operator, there is no way of deciding which representation should be selected for outputting the term since output in prefix notation and output in postfix notation are both correct.

The operator declarations are the same as in the previous example. The following call can therefore produce one of the above outputs.

```
[user] ?- write(die(say(never))).

die say never
```

If a left- and a right-associative operator have the same precedence, then ambiguities may arise when a term containing both operators is read. Such ambiguities are always resolved in favor of the right-associative operator, i.e. in this case the binding of the right-associative operator is stronger. Such operator declarations should be avoided, however.

9.3.2 List of built-in operators

The following table is an overview of the built-in operators.

Operator	Prec.	Type	Pred./Func.	Meaning
`:-`	1200	xfx		Separates clause head and body of a clause
`-->`		xfx		Generation of syntax rules
`:-`		fx		Identifies a directive
`?-`		fx		Prompt
`;`	1100	xfy	Pred.	Disjunction of goals
`->`	1050	xfy	Pred.	Conditional ("if -> then")
`','`	1000	xfy	Pred.	Conjunction of goals
`not`	900	fy	Pred.	Negation through failure
`\+`	900	fy	Pred.	Negation through failure
`=>`	900	xfx		Context info
`is`	700	xfx	Pred.	Evaluate arithmetic expression
`=..`		xfx	Pred.	Convert between a structure and a list
`@>=`		xfx	Pred.	Comparison predicates
`@=<`		xfx	Pred.	for terms
`@>`		xfx	Pred.	
`@<`		xfx	Pred.	
`\==`		xfx	Pred.	
`==`		xfx	Pred.	
`>=`		xfx	Pred.	Comparison predicates
`=<`		xfx	Pred.	for numeric values
`>`		xfx	Pred.	
`<`		xfx	Pred.	
`=\=`		xfx	Pred.	
`=:=`		xfx	Pred.	
`\=`		xfx	Pred.	Test for non-unifiability
`=`		xfx	Pred.	Unification
`\/`	500	yfx	Func.	Logical OR
`/\`		yfx	Func.	Logical AND
`#`		yfx	Func.	Logical XOR
`+`		yfx	Func.	Addition
`-`		yfx	Func.	Subtraction
`/`	400	yfx	Func.	Division
`//`		yfx	Func.	Integer division
`*`		yfx	Func.	Multiplication
`rem`		yfx	Func.	Remainder
`mod`		yfx	Func.	Modulo
`>>`		yfx	Func.	Binary shift right
`<<`		yfx	Func.	Binary shift left
`\`	200	fx	Func.	Logical complement
`**`		yfx	Func.	Exponentiation
`+`		fx	Func.	Neutral function
`-`		fx	Func.	Negation
`@`	100	xfx		Module specification
`:`	50	xfx		Definition module of the predicate

Some of the operators in this table have special meanings when the Prolog system is reading in clauses. The remaining operators are functors of predicates and arithmetic functions.

9.3.3 Test and definition predicates

Predicate	Task
op/3	Define or delete operators
current_op/3	Query information about operator definitions

Operators are declared with `op(` *Precedence*`,` *Type*`,` *Names*`)`. `op/3` can be used as a predicate or as a directive. An existing operator definition is overwritten by a new operator definition. If 0 is specified as the precedence, an existing operator definition is deleted.

9.4 Arithmetic expressions and functions

An arithmetic expression is a structure whose functors are all arithmetic functions. The arguments of these functions are numbers, arithmetic functions or arithmetic expressions.

Examples

```
N + 1
2.2 * K + 33.9
time // 1000
```

These expressions are evaluated in the following cases:

- if the expression occurs as the second argument of `is`
  `(X is arithmetic_expression)`
- if arguments in a Prolog predicate are evaluated implicitly (e.g. the comparison predicates for numeric values).

The evaluation sequence is determined by the type and precedence of the operators involved, and by any explicit parentheses. The order of evaluation conforms to the usual mathematical conventions (unless the operator definitions are modified).

A list of arithmetic functions can be found in the description of the `is/2` predicate in this manual.

The available functions can be divided into two classes: arithmetic functions and logical functions.
The value of an arithmetic function results from the value of the argument(s) based on normal mathematical conventions.
The logical functions are defined only for integer arguments. The corresponding bits in the

internal representation of these integers are ANDed and ORed as specified: the resulting bit pattern again represents an integer, which is the result of the function.

Many standard arithmetic functions (trigonometry, logarithms, etc.) are available as Prolog functions.

Examples of arithmetic expressions

```
[user] ?- 5=4+1.                % no evaluation, term unification

no

[user] ?- N = 4, N1 is N + 1.

N       = 4
N1      = 5

yes

[user] ?- K = 33, T is 2.2 * K + 33.9.

K       = 33
T       = 106.5

yes

[user] ?- X is time // 1000.

X       = 709476

yes
```

9.5 Prolog text

A Prolog text consists of clauses, directives and grammar rules.

Clauses define predicates which can be stored in the data base and be executed. All built in predicates and language constructs are described in greater detail in the alphabetic section of this manual.

Directives are evaluated when Prolog programs are loaded; They are used to control loading, to structure the Prolog text in modules or to declare properties of Prolog objects. All directives are described in greater detail in the alphabetic section of this manual.

Grammar rules are also clauses which provide a special syntactical form to make work with grammars easier (see *IF/Prolog User's Guide* [2]).

Clause

$$\left\{ \begin{array}{l} literal\ . \\ literal\ :-\ goal\ . \end{array} \right\}$$

Directive

$$:-\ literal\ .$$

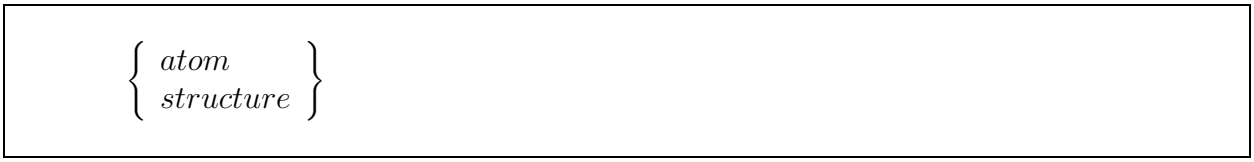
Grammar rule

$$\left\{ \begin{array}{l} literal\ -->\ production\ . \\ literal\ ,\ terminals\ -->\ production\ . \end{array} \right\}$$

Goal

$$\left\{ \begin{array}{l} variable \\ literal \\ moduleName\ :\ literal \\ literal\ @\ module \\ moduleName\ :\ literal\ @\ module \\ goal\ ,\ goal \\ goal\ ;\ goal \\ goal\ ->\ goal \\ goal\ ->\ goal; goal \\ \texttt{call(}\ goal\ \texttt{)} \\ \texttt{call(}\ goal\ \texttt{)}@module \\ \texttt{not}\ goal \\ \backslash +\ goal \\ \texttt{true} \\ \texttt{fail} \\ ! \end{array} \right\}$$

Literal



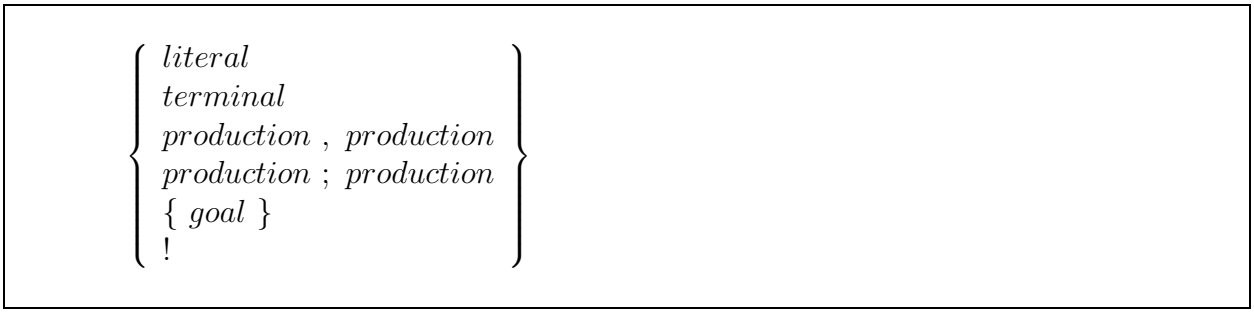
Module



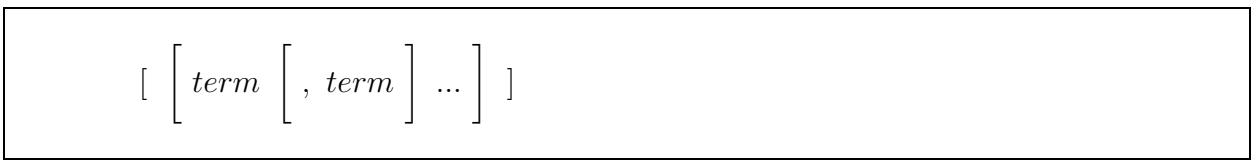
ModuleName



Production



Terminals



Appendix A

ISO Standard Directives and Predicates

Following directives are part of the ISO Standard:

char_conversion/2	discontiguous/1	dynamic/1
ensure_loaded/1	include/1	initialization/1
multifile/1	op/3	set_prolog_flag/2

Following control constructs are part of the ISO Standard:

!/0	,/2	;/2	->/2
call/1	catch/3	fail/0	throw/1
true/0			

Following predicates are part of the ISO Standard:

=/2	=../2	==/2
\=/2	\+/2	\==/2
=:=/2	=\=/2	@=</2
@</2	@>=/2	@>/2
=</2	</2	>=/2
>/2	abolish/1	arg/3
at_end_of_stream/0/1	atom/1	atom_chars/2
atom_codes/2	atom_concat/3	atom_length/2
atomic/1	asserta/1	assertz/1
bagof/3	char_code/2	char_conversion/2
clause/2	close/1/2	compound/1
copy_term/2	current_char_conversion/2	current_input/1
current_op/3	current_output/1	current_predicate/1
current_prolog_flag/2	findall/3	float/1
flush_output/0/1	functor/3	get_byte/1/2
get_char/1/2	get_code/1/2	halt/0/1
integer/1	is/2	nl/0/1
nonvar/1	number/1	number_chars/2
number_codes/2	once/1	op/3
open/3/4	peek_byte/1/2	peek_char/1/2
peek_code/1/2	put_byte/1/2	put_char/1/2
put_code/1/2	read/1/2	read_term/2/3
repeat/0	retract/1	set_input/1
set_output/1	set_prolog_flag/2	set_stream_position/2
setof/3	stream_property/2	sub_atom/5
unify_with_occurs_check/2	var/1	write/1/2
writeq/1/2	write_canonical/1/2	write_term/2/3

Appendix B

ASCII table

Decimal	Octal	Hexadec.		Meaning	Control
0	00	00	NUL	Null, no operation	@
1	01	01	SOH	Start of Heading	A
2	02	02	STX	Start of Text	B
3	03	03	ETX	End of Text	C
4	04	04	EOT	End of Transmission, END key	D
5	05	05	ENQ	Enquiry	E
6	06	06	ACK	Acknowledge	F
7	07	07	BEL	Bell	G
8	10	08	BS	Backspace	H
9	11	09	HT	Horizontal Tabulation	I
10	12	0A	LF	Line Feed	J
11	13	0B	VT	Vertical Tabulation	K
12	14	0C	FF	Form Feed	L
13	15	0D	CR	Carriage Return	M
14	16	0E	SO	Shift Out	N
15	17	0F	SI	Shift In	O
16	20	10	DLE	Data Link Escape	P
17	21	11	DC1	Device Control 1, Continue output	Q
18	22	12	DC2	Device Control 2	R
19	23	13	DC3	Device Control 3, Stop output	S
20	24	14	DC4	Device Control 4	T
21	25	15	NAK	Negative Acknowledge	U
22	26	16	SYN	Synchronous Idle	V
23	27	17	ETB	End of Transmission Block	W
24	30	18	CAN	Cancel	X
25	31	19	EM	End of Medium	Y
26	32	1A	SUB	Substitute Character	Z
27	33	1B	ESC	Escape	
28	34	1C	FS	File Separator	
29	35	1D	GS	Group Separator	]
30	36	1E	RS	Record Separator	
31	37	1F	US	Unit Separator	-

Decimal	Octal	Hexadec.		Meaning
32	40	20	SP	SPACE
33	41	21	!	
34	42	22	”	
35	43	23	#	Number symbol
36	44	24	$	or national currency symbol
37	45	25	%	
38	46	26	&	
39	47	27	’	
40	50	28	(	
41	51	29	)	
42	52	2A	⋆	Asterisk, multiplication sign)
43	53	2B	+	
44	54	2C	,	
45	55	2D	-	
46	56	2E	.	
47	57	2F	/	Division sign
48	60	30	0	
49	61	31	1	
50	62	32	2	
51	63	33	3	
52	64	34	4	
53	65	35	5	
54	66	36	6	
55	67	37	7	
56	70	38	8	
57	71	39	9	
58	72	3A	:	
59	73	3B	;	
60	74	3C	<	
61	75	3D	=	
62	76	3E	>	
63	77	3F	?	

Decimal	Octal	Hexadec.		Meaning
64	100	40	@	(commercial "at" or $)
65	101	41	A	
66	102	42	B	
67	103	43	C	
68	104	44	D	
69	105	45	E	
70	106	46	F	
71	107	47	G	
72	110	48	H	
73	111	49	I	
74	112	4A	J	
75	113	4B	K	
76	114	4C	L	
77	115	4D	M	
78	116	4E	N	
79	117	4F	O	
80	120	50	P	
81	121	51	Q	
82	122	52	R	
83	123	53	S	
84	124	54	T	
85	125	55	U	
86	126	56	V	
87	127	57	W	
88	130	58	X	
89	131	59	Y	
90	132	5A	Z	
91	133	5B	[	
92	134	5C	\	Backslash
93	135	5D	]	
94	136	5E	~	
95	137	5F	_	Underscore

Decimal	Octal	Hexadec.		Meaning
96	140	60	‘	
97	141	61	a	
98	142	62	b	
99	143	63	c	
100	144	64	d	
101	145	65	e	
102	146	66	f	
103	147	67	g	
104	150	68	h	
105	151	69	i	
106	152	6A	j	
107	153	6B	k	
108	154	6C	l	
109	155	6D	m	
110	156	6E	n	
111	157	6F	o	
112	160	70	p	
113	161	71	q	
114	162	72	r	
115	163	73	s	
116	164	74	t	
117	165	75	u	
118	166	76	v	
119	167	77	w	
120	170	78	x	
121	171	79	y	
122	172	7A	z	
123	173	7B	{	
124	174	7C	|	
125	175	7D	}	
126	176	7E		
127	177	7F	DEL	Delete Interrupt (Signal)

Appendix C

Glossary

alias name: Atom associated with an open stream.

argument: An object in a Prolog structure. In the term `mother(gaia, uranus)`, for example, `gaia` and `uranus` are the arguments.

arity: A non-negative integer, representing the number of arguments in a structure.

atom: A basic object incapable of further subdivision, denoted by an identifier.

backtracking: Proving method which in the event of a failed route in a blind alley returns to the last choice point in the proof tree and checks alternative paths.

body: Condition, righthand part of a rule.

box model: Model illustrating how Prolog proves a goal.

breadth-first search: Search strategy in proof trees which considers several paths simultaneously.

built-in predicate: A predicate defined by the Prolog system.

choice point: Node in the proof tree at which multiple possibilities to satisfy or refute the goal exist.

clause: Statement about objects and relations concerning objects; a fact or a rule.

compound term: see structure

conjunction (,): Combination of statements. A statement `A , B` is true when both statements `A` and then `B` are true.

constant: An atom or a number.

constraint: Restriction for the value of one or more variables.

consult: To read in, analyse and store in the database a Prolog text, executing any directives.

context: Logical environment for the entire proof tree covered by the specified goal.

cut: A built-in predicate that removes choice points.

database: Set of predicates and all the objects managed by Prolog (excluding Prolog stacks).

declaration part (of a module): Is defined by the directives which are possible in it.

depth-first search: Search strategy in proof trees which continues to pursue only one path until that path is successful or ends in a blind alley. In the latter case, for example, backtracking can be initiated.

directive: Structure having the functor `:-` and arity 1. A directive is evaluated when Prolog text is read in.

disjunction (;): ORing of statements. A statement `A ; B` is true when at least one of the statements `A` or `B` is true.

dynamic predicate: Predicate whose clauses may be be modified during execution.

error context: Context in which an error is handled.

exception: Programmable event. In the event of user errors the system branches to the part of the program defined to handle this exception. If the exception is not caught, the system issues an error message.

export: Making a predicate visible to, and thus usable by, other modules.

expression: A term consisting of numbers and compound terms with evaluable functors.

extended term: Combination of terms using operators.

fact: Fact concerning the properties and names of objects. A fact describes a clause whose body is `true`.

flag: see Prolog flag

functor: The name of a predicate or structure.

global variable: Variable whose scope is not local to a clause and whose value is not lost during backtracking.

goal: Term whose proof is attempted with depth-first searching, backtracking and unification.

ground: A ground term contains no variables.

head: Conclusion, lefthand part of a clause.

implementation part (of a module): Contains predicate and operator definitions.

import: Making a predicate from another module visible and thus usable.

infix notation: Structure notation where the operator is located between the operands, e.g. `(A+B)`.

instantiated variable: Variable which is instantiated with a constant or a compound term.

instantiation: Substitution of a variable by a non-variable.

list: A Prolog term, either the empty list `[]` or a compound term whose functor is `'.'` and which has two arguments, the second of which is a list.

literal: Atom or structure.

load: Reading compiled modules into the database.

local variable: Variable which is visible in only one clause.

logical variable: A term which can represent any term.

metapredicate: A metapredicate has an additional argument which specifies the module in whose context it is to be executed.

module: Logical collection of predicates; a module has a module interface and a module body which are interlinked by a common module name.

module body: Private portion of the module, introduced by the `begin_module` directive and containing a declaration part and an implementation part.

module interface: Public portion of the module, introduced by the `module` directive.

operator: Atom which may occur as a functor in prefix, infix or postfix notation. The notation of the operator is determined by the predicate or the directive `op/3`. An operator has an precedence and an associativity which make it possible to determine which arguments and terms of different or matching precedence can be assigned to one another.

overflow: A calculated value is greater than the storage location provided for it.

parser: Component of the Prolog system which converts character strings into Prolog terms.

postfix notation: Structure notation where the operator follows the operands, e.g. `(A,B)-`.

predicate: A relation identified by functor and arity.

prefix notation: Structure notation where the operator precedes the operands, e.g. `+(A,B)`.

predicate indicator: A compound term *Functor*/*Arity*.

principal functor: Outermost functor in term notation.

procedure: An empty or non-empty sequence of clauses in a database defining a predicate.

Prolog flag: System state which can be queried and in some cases set by the user.

Prolog text: A sequence of terms, representing clauses and directives.

proof: Sequence of logical conclusions which produce a statement from facts.

proof tree: The paths taken by the system during proving of a goal can be mapped onto a tree structure; both the successful paths and also the blind alleys are included in the tree.

qualification: Notation specifying the module in which it was defined.

query: Request to the Prolog system to prove one or more goals.

recursion: Relation or object which is based on itself.

resolution: The proof method of Prolog.

rule: Statement in the form `If A, then B`, term in the database in the form `B:-A`, where A is the rule body and B the rule head.

side effect: A non-logical effect of a built-in predicate, such as input, output, arithmetic evaluation or database modification.

stack: Type of memory usage; the values are pushed onto the stack and the last value stored is the first value processed again.

statement: Description of a situation which can be either exactly true or

exactly false.

static predicate: Predicate which may not be modified.

stream: Unique designation for an open input/output channel to a file, terminal or printer, for example. This designation is created by the Prolog system and can also be referenced by means of a user-defined alias name.

structure: Term consisting of a functor and a sequence of arguments. The number of arguments is the arity of the term. A structure is written in functor notation as follows: functor, followed without intervening space by an opening left parenthesis, followed by comma-separated arguments, and a closing right parenthesis. If the arity is zero, the term is also referred to as an atom and is written without parentheses.
For structures with an arity of one or two an alternative notation may be used if the functor has been defined as a prefix, infix or postfix operator.

subgoal: Term whose proof must succeed in order that the given goal may be proved.

term: One of the objects: variable, number, atom or structure.

trail stack: Memory area in the Prolog system for temporary storage of variable instantiations.

user predicate: Predicate which must be defined by the user.

unification: Method for making two terms identical through the substitution of variables.

Bibliography

[1] IF/Prolog V5.3 Reference Manual

[2] IF/Prolog V5.3 User's Guide

[3] IF/Prolog V5.3 OSF/Motif Interface

[4] IF/Prolog V5.3 Informix Interface

[5] IF/Prolog V5.3 Constraints Package

[6] IF/Prolog V5.3 Quick Reference

[7] IF/Prolog V5.3 Windows Interfaces

[8] IF/Prolog V5.3 Java Interface

[9] IF/Prolog V5.3 BDD Package

[10] X/Open CAE (Common Applications Environment) Specification. System Interfaces and Headers, Issue 4. Prentice Hall 1994.

[11] International Standard, ISO/IEC IS 13211-1. International Organization for Standardization, 1995

[12] William F. Clocksin, Chris S. Mellish: Programming in PROLOG. Standard Edition Berlin et al.: Springer 1995.

[13] Ivan Bratko: PROLOG. Programming for Artificial Intelligence Second Edition, Addison-Wesley 1990.

[14] Leon Sterling, Ehud Shapiro: The Art of PROLOG. Advanced Programming Techniques. Cambridge Massachusetts: MIT Press, 1986

Index

abs, 309
absolute value, 309
access structure arguments, 740
acos, 309
add a complex C function, 693
add a simple C function, 694
addition, 309
anonymous variable, 806
appending lists, 77
arc cosine, 309
arc sine, 309
arc tangent, 309
argument
    precedence
        associativity, 814
arithmetic expression, 820
arithmetic function, 309, 820
arity, 809
array
    create, 171
    dimension, 83
    query element, 265
    set element, 477
array term
    test, 82
ASCII character
    syntax, 802
asin, 309
associativity of an argument, 814
atan, 309
atom
    analyze, 545
    compose, 103
    concatenation, 159
    concatenation of atoms, 341
    convert to character code list, 101
    convert to character list, 99
    convert to number, 107
    decompose, 103, 113
    generate, 764
    index, 304
    number of bytes, 124
    number of characters, 105
    parse, 392
    prefix, 111
    subatom, 109, 545
    suffix, 115
    syntax, 799
    syntax analysis, 392
    test, 97, 749
    unify, 781

backslash, 799
backtracking
    disable, 39
    initiate, 243
binary number
    syntax, 802
bitwise conjunction, 309
bitwise disjunction, 309
bitwise exclusive or, 309
bitwise left shift, 309
bitwise right shift, 309
body of a module, 121
Branch back to CALL port of parent goal, 635
Branch back to CALL port of subgoal, 636
Branch back to clause head, 634
Breakpoint reactivate, 632
buffer
    flush, 736
byte
    input, 266, 395
    output, 428

C Function
    call on IF/Prolog shutdown, 695

link to IF/Prolog, 692
C predicates
argument access, 716
call
grammar rules, 401
call a goal automatically, 420
CALL port, 635, 636
ceiling, 309
character
access in atom, 283
ASCII, 802
input, 268, 271, 397, 399
output, 430, 432
character code, 133
character code list
convert to atom, 101
convert to number, 374
character conversion
define, 135, 136
query, 174
character list
convert to atom, 99
convert to number, 371
character set, 793
character string
read, 281
characters
meaning of Prolog-specific, 794
Prolog-specific, 794
special, 794
close a Prolog list, 741
command
execute, 552, 554
command interpreter
start, 550
comments, 796
end-of-line, 796
variable-length, 796
compiler, 155
complement, 309
complex C function
add to a module, 708
configure
debugger, 208
tracer, 576
conjunction, 309
constant
concatenation, 159
syntax, 799
test, 117
constant pi, 309
context
create, 166
quit, 568
context jump
initiate, 568
intercept, 131
control construct, 125, 127, 243, 580
conjunction, 42
cut, 39
disjunction, 53
IF-THEN conjunction, 47
negation, 71, 368
cos, 309
cosh, 309
cosine, 309
CPU time, 309
cputime, 309
create
array, 171
csize, 309
current directory
change, 138
query, 285
current module
define, 479
query, 175
cursor
position, 356, 389, 563
cused, 309
cut, 39
cyclic term
test, 204

database size, 309
database used, 309
date
determine, 332
Deactivate breakpoint temporarily, 640
debug
goal, 205

debugger
configure, 208
initialize, 207
termination, 211
declaration of predicates, 413
decomposing a list, 77
delimiters, 794
in expressions, 796
in structures, 796
device driver
register, 696
digit
test, 218
dimension
array, 83
directive
begin_module, 121
char_conversion, 136
discontiguous, 219
dynamic, 221
end_module, 225
ensure_loaded, 227
export, 241
import, 299
include, 302
init, 306
initialize, 306
meta, 349
module, 354
multifile, 358
nonotify, 366
op, 383
private, 413
reexport, 450
set_prolog_flag, 493
directory
change current, 138
check access permission, 245
check existence, 245
query current, 285
disabling the return key, 797
disjunction, 309
Display breakpoints, 679
Display constraints, 674
Display default values, 673
Display fields
select, 671
Display help information, 647
Display history, 678
Display list of predicates in module, 675
Display listing of a predicate, 676
Display variable constraints, 680
division, 309
division remainder, 309
dsize, 309
dused, 309

editor
call, 223
query, 215
set, 215
element
array, 265, 477
individual list element, 324
last list element, 317
list membership, 344, 347, 364
of a list, 812
empty list
generate, 773
test, 760
unify, 789
end of file
query, 95
end of line
query, 93
end of stream
query, 95
end-of-line comments, 796
environment
change drive, 599
file information, 273
query drive, 600
environment variable
query, 287
epilog
enable, 719
error context
clear, 698
query, 700
set, 703
status, 701

esize, 309
eused, 309
evaluate expression, 753
exception
    handle, 166, 236
    intercept, 131, 166
    raise, 233
Execute a Prolog goal, 638
exp, 309
exponentiation, 309
export predicates, 241, 349
expression
    arithmetic, 820
    compare, 55
    compute, 55
    converting to normal structure, 813
    evaluate, 309
    syntax, 810
extended term stack
    size, 309
    used, 309

FAIL port, 644
FAILBODY port, 644
FAILMATCH port, 644
file
    check access permission, 245
    check existence, 245
    edit, 223
file information, 273, 599, 600
fixed-point number
    syntax, 803
float, 309
floating-point expression
    test, 753
floating-point number
    generate, 767
    query format, 253
    set format, 253
    syntax, 803
    test, 252, 752
    unify, 784
float_fractional_part, 309
float_integer_part, 309
floor, 309
Force current subgoal to succeed, 643
Force the current subgoal to fail, 644
function, 820
    arithmetic, 309, 820
    logical, 309, 820

garbage collection
    activate, 264
global stack
    size, 309
    used, 309
global variable
    define, 427, 482, 807
    delete, 595
    name, 806
    pop value, 403
    push value, 427
    query names, 179
    query value, 276, 403
    set value, 482
    value, 807
goal
    all solutions, 118, 249, 499
    call, 125, 127
    call automatically, 420
    debug, 205
    profile, 415
    trace, 573
    true, 243, 580
goals
    conjunction, 42
    disjunction, 53
    IF-THEN conjunction, 47
    IF–THEN–ELSE conjunction, 296
grammar rules
    call, 401
    consult, 163
    expand, 240
    operator, 44
    reconsult, 446
    transform, 240
gsize, 309
gused, 309

head
    of a list, 812
hexadecimal number

syntax, 802
history
configure, 654

import predicates, 299
infix operator, 813
input
byte, 266, 395
character, 268, 271, 397, 399
character string, 281
query end of line, 93
query end of stream, 95
skip line, 507
syntax errors, 548
term, 437, 442
input loop
generate, 123
installation path
of IF/Prolog system, 426
instantiation
test, 289, 367
integer
generate, 770
syntax, 801
test, 308, 756
unify, 787
integer division, 309
integer expression
test, 757
integers
generate a sequence, 258
interface of a module, 354
invocation parameters
query, 422
iteration
for, 258
repeat, 456

left shift, 309
letter
test, 316
lexical elements, 795
line
skip, 507
list
append, 77
convert to structure, 60
decompose, 77
element, 812
form from head and tail, 161
generate, 771, 777
head, 812
individual element, 324
last element, 317
number of elements, 319
remove sublist, 454
reverse, 463
sort, 527
syntax, 811
tail, 812
test, 758
test membership, 344, 347, 364
unify, 788
literal, 821
local (control) stack
size, 309
used, 309
log, 309
logical function, 309, 820
loop
for, 258
repeat, 456
lowercase letters
convert to uppercase, 334

machine identification
query, 558
match
pattern, 338, 341
regular expression, 452
maximum function, 309
maxint, 309
memory
database, 309
global stack, 309
local (control)stack, 309
stacks, 309
trail stack, 309
memory management
query parameters, 183
set parameters, 484
statistics, 529

metapredicate
  declare, 349
  query module context, 129
minimum function, 309
minint, 309
mod, 309
module
  add a complex C function, 708
  add a simple C function, 709
  compile, 155
  conclude definition, 225
  declare metapredicates, 349
  declare predicates, 413
  define, 354
  define body, 121
  define current, 479
  define interface, 354
  delete, 594
  explicit specification, 52
  export predicates, 241
  import predicates, 299
  initialize, 306
  load, 155, 328
  query current, 175
  query defined, 184
  reexport predicates, 450
  set call context, 70
  store in file, 465
modulo function, 309
multiplication, 309

negation, 71, 368
newline
  output, 362
normal structure, 809
notational conventions, 3
number
  binary, 802
  convert to atom, 107
  convert to character list, 371
  convert to list of character codes, 374
  fixed-point, 803
  floating-point, 803
  hexadecimal, 802
  octal, 802
  rational, 804
  syntax, 801
  test, 370
numerical value
  calculate, 309

occurs check, 583
octal number
  syntax, 802
operating system
  change drive, 599
  file information, 273
  query, 560
  query drive, 600
operator
  define, 379, 383
  delete, 379, 383
  infix, 813
  postfix, 814
  precedence, 814
  prefix, 813
  query, 186
  syntax, 813
  type, 813
ordering of terms, 65, 152
output
  byte, 428
  character, 430, 432
  newline, 362
  position cursor, 356, 389, 563
  readable format, 605
  term, 411, 601, 607, 613, 618
  user-defined, 404

parameters
  invocation, 422
  system, 561
  user-defined, 596
pattern matching, 338, 341
pipe
  create, 590
position of a character string in an atom, 304
postfix operator, 814
precedence
  of an operator, 814
predicate

declare, 413
export, 241
IF-THEN-ELSE conjunction, 296
import, 299
reexport, 450
prefix
atom, 111
prefix operator, 813
process
create, 584
fork, 584
query ID, 587
send signals, 588
spawn, 584
wait for termination of child, 592
production operator, 44
profile
goal, 415
query statistics, 418
reset statistics, 417
Prolog compiler
call, 7
Prolog cross-referencer
call, 17
Prolog definition referencer
call, 8
Prolog flag
query, 191
set, 488, 493
Prolog optimizer
call, 13
Prolog preprocessor
call, 14
Prolog system
call, 9
Prolog tag generator
call, 16
Prolog text
compile, 155
load, 50, 163, 328, 446
Prolog-specific characters, 794
meaning of, 794

Raise an exception, 642
random, 309
random number, 309
rational division, 309
rational number
decompose, 435
denominator, 435
numerator, 435
syntax, 804
test, 434
rdiv, 309
read
term, 437, 442
Reconfigure Prolog, 6
reexport predicates, 450
reference
backward-chained, 17
forward-chained, 17
regular expression
match, 452
rem, 309
Remove explicit breakpoints, 651
return key
delimiter function, 796
disabling, 797
reverse
list, 463
right shift, 309
round, 309

screen
clear, 148
clear line, 144
clear to end, 146
determine cursor position, 277
query size, 581
Scrolling, 630
separators, 795
sequence of integers
generate, 258
serial number
of IF/Prolog system, 424
Set debugger port, 649
Set explicit breakpoint, 661
Set length of trace field, 656
Set structure depth, 653
Set structure depth for trace outputs, 655
Set trace ports, 668
shell

start, 550
Show dynamic call chain, 672
sign, 309
signal
control, 502
handle, 166, 505
query, 197
send to process, 588
signum, 309
simple C function
add to a module, 709
sin, 309
sine, 309
sinh, 309
size
database, 309
global stack, 309
local (control)stack, 309
stacks, 309
trail stack, 309
sort a list, 527
special characters, 794
sqrt, 309
square root, 309
ssize, 309
stack
size, 309
used, 309
Start new input loop, 637
start shell, 666
statistics
profile, 417, 418
Store field, 658
stream
input line, 247
position of line, 247
redefine, 535
string
syntax analysis, 392
structure
access arguments, 80
arity, 809
convert to list, 60
determine arity, 261
determine functor, 261
generate, 261, 765, 768, 774
normal, 809
syntax, 808
test, 157, 750, 754, 761
unify, 782, 785, 790
sublist
remove from a list, 454
subtraction, 309
suffix
atom, 115
sused, 309
Switch off trace mode, 667
Switch on trace mode, 667
syntax
notational conventions, 3
syntax analysis
string, 392
syntax error, 440
system parameters
query, 561
system status
load, 330
save, 467, 469
system time, 309

tags, 16
tail
of a list, 812
tan, 309
tangent, 309
tanh, 309
term
comparison, 63, 65
convert into atom, 603, 620
copy, 170
formatted output, 607, 611
input, 437, 442
last list element, 317
list element, 324
member of list, 344, 347, 364
ordering, 65
output, 411, 601, 605, 613
output in readable form, 618
read, 437, 442
syntax, 798
test non-unifiability, 68

type, 746, 778
unification, 58, 583, 780, 792
term context
determine, 745
release term variables, 742
term manipulation
add to a Prolog list, 738
term variables
release, 742
Terminate debugger and IF/Prolog, 646
Terminate interactive debugger, 631
terms
comparison, 152
ordering, 152
test
for array term, 82
for atom, 97
for constant, 117
for cyclic term, 204
for digit, 218
for floating-point number, 252
for instantiation, 289, 367
for integer, 308
for letter, 316
for number, 370
for rational number, 434
for structure, 157
for variable, 598
time, 309
determine, 332, 570
timezone
determine, 570
toplevel
generate, 123
return to, 75
trace
goal, 573
tracer
configure, 576
initialize, 575
output predicate, 579
termination, 578
trail stack
size, 309
used, 309
transform term
grammar rules, 240
truncate, 309
TRYMATCH port, 634
tsize, 309
tused, 309
type
operator, 813

undo unification, 730
unification
term, 58, 583
term with atom, 781
term with floating-point number, 784
term with integer, 787
term with list, 788
term with structure, 782, 785, 790
term with term, 780, 792
term with the empty list, 789
undo, 730
with occurs check, 583
Unify variable, 670
univ, 60
uppercase letters
convert to lowercase, 334
user parameters
query, 596

variable
anonymous, 806
free, 118, 499
generate, 776
global, 806
single-variable check, 806
syntax, 804
test, 598, 763
variable name, 805
version
of IF/Prolog system, 425

wait
for end of process, 592